DOCUMENT RESUME

ED 263 219 TM 850 721

AUTHOR	Pennington, Nancy
TITLE	Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs.
INSTITUTION	Chicago Univ., Ill. Graduate School of Business.
SPONS AGENCY	Office of Naval Research, Arlington, Va. Personnel and Training Research Programs Office.
REPORT NO	TR-2-ONR
PUB DATE	Jan 85
CONTRACT	N00014-82-K-0759
NOTE	92p.; A product of the Center for Decision Research.
PUB TYPE	Reports - Research/Technical (143)
EDRS PRICE	MF01/PC04 Plus Postage.
DESCRIPTORS	Adults; *Cognitive Ability; Cognitive Processes; Cognitive Style; *Comprehension; Computer Assisted Testing; Computer Software; Language Processing; Memory; *Problem Solving; Programers; *Programing; *Programing Languages; Schemata (Cognition)

ABSTRACT

The difficulty of computer programming and other design tasks is a function of the multiple abstractions that combine to form a program's meaning. Thus, comprehension of the final form of a program design, the program text, involves retrieving multiple sets of relationships between parts. The research presented here addresses which, if any, abstraction is dominant psychologically in programmers' mental representations of programs, specifically whether procedural (control flow) or functional (goal hierarchy) abstractions dominate, and the relative ease or difficulty of inferring different kinds of information from the text. Eighty professional programmers were tested on comprehension and recognition of short program texts. The results, which suggest that procedural rather than functional units form the basis of expert programmers' mental representations, support work in other areas of text comprehension showing the importance of text structure knowledge in understanding. The results also raise questions concerning claims that "chunking" found in comprehension and memory tasks in skilled problem solving domains is due to pattern matching to schematic content knowledge rather than to knowledge of more abstract text or design sturcture units. A reference list, 4 tables, and 10 figures are appended. (Author/LMO)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED263219

CENTER FOR DECISION RESEARCH

Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs

Nancy Pennington
Center for Decision Research
Graduate School of Business
University of Chicago

January 1985

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☑ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

TM 860 721

Graduate School of Business
The University of Chicago

Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs

Nancy Pennington
Center for Decision Research
Graduate School of Business
University of Chicago

January 1985

Running Head: Representations in Comprehension

This research was sponsored by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract N00014-82-K-0759, Contract Authority Identification Number 667-503. Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government.

Beatrice Grabowski, Paul Harvell, Lori Hunsaker, John Keating and Helena Szepe worked on the project as research assistants. Thanks are due to Sargent and Lundy, and Argonne Laboratories for encouraging their programmmers to participate in the research, and to many other professional programmers who came to our lab to participate. Thanks are also due to Reid Hastie and Beatrice Grabowski for helpful comments on the manuscript. Correspondence should be directed to Nancy Pennington, Center for Decision Research, Graduate School of Business, University of Chicago, 1101 E. 58th Street, Chicago, IL 60637.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) Technical Report No. 2-ONR	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Graduate School of Business University of Chicago 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Personnel and Training Research Group Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 East 58th Street Chicago, IL 60637	7b. ADDRESS (City, State, and ZIP Code) Code 442 PT Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-82-K-0759
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
61153N	RR042-06	RR042-06-0A	NR667-503

11. TITLE (Include Security Classification)

Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs

12. PERSONAL AUTHOR(S)
Pennington, Nancy

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical Report	FROM 9/82 TO 1/85	January 1985	105

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	computer programming, expertise, text comprehension, cognitive skill, problem solving

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The difficulty of computer programming and other design tasks is a function of the multiple abstractions that combine to form a program's meaning. Thus, comprehension of the final form of a program design, the program text, involves retrieving multiple sets of relationships between parts. The research presented here addresses which, if any, abstraction is dominant psychologically in programmers' mental representations of programs, specifically whether procedural (control flow) or functional (goal hierarchy) abstractions dominate, and the relative ease or difficulty of inferring different kinds of information from the text. Eighty professional programmers were tested on comprehension and recognition of short program texts. The results, which suggest that procedural rather than functional

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

units form the basis of expert programmers' mental representations, support work in other areas of text comprehension showing the importance of text structure knowledge in understanding. The results also raise questions concerning claims that "chunking" found in comprehension and memory tasks in skilled problem solving domains is due to pattern matching to schematic content knowledge rather than to knowledge of more abstract or design structure units.

ABSTRACT

The difficulty of computer programming and other design tasks is a function of the multiple abstractions that combine to form a program's meaning. Thus, comprehension of the final form of a program design, the program text, involves retrieving multiple sets of relationships between parts. The research presented here addresses which, if any, abstraction is dominant psychologically in programmers' mental representations of programs, specifically whether procedural (control flow) or functional (goal hierarchy) abstractions dominate, and the relative ease or difficulty of inferring different kinds of information from the text. Eighty professional programmers were tested on comprehension and recognition of short program texts. The results, which suggest that procedural rather than functional units form the basis of expert programmers' mental representations, support work in other areas of text comprehension showing the importance of text structure knowledge in understanding. The results also raise questions concerning claims that "chunking" found in comprehension and memory tasks in skilled problem solving domains is due to pattern matching to schematic content knowledge rather than to knowledge of more abstract text or design structure units.

The goal of a problem solver is to piece together a collection of actions that will achieve a set of goals. This description is particularly applicable to a large and varied subgroup of problem solving tasks that have been designated variously as design tasks (Greeno & Simon, 1984), engineering tasks (Shrobe, 1979), or composition tasks (Reitman, 1965). Examples of design tasks include house design, electrical circuit design, composing a piece of music, choreographing a dance, writing an essay, writing an instruction manual, and writing computer programs, to name a few. In fact, all problem solving tasks may be viewed as design tasks when conceptualized in a certain way (see Greeno & Simon, 1984) but the distinctive feature of the task domains under consideration here is that the immediate goal of the problem solver is to design a complete problem solution that someone else will interpret and execute, rather than to achieve the specified goals of the problem directly. To illustrate this distinction, consider the task of solving a cryptarithmetic problem, a puzzle problem that has been used in problem solving research (Newell & Simon, 1972). In cryptarithmetic problems, the problem solver is presented with an arithmetic problem complete with answer in which the numbers have been replaced by letters. The goal is to determine an assignment of digits to letters so that the numerical operands combine properly to yield the answer. When this problem is solved, numbers are assigned to letters and the goals of the problem are satisfied directly. However the sequence of actions that produced the goal state is lost. In contrast, consider the problem of writing a computer program that will solve exactly the same (single) cryptarithmetic problem (or a class of cryptarithmetic problems). Here the immediate goal is to lay out the sequence of actions in advance of their execution rather than to achieve the goal immediately. Indeed it is the computer in this case that will achieve the goal of assigning numbers to

letters by executing the programmer's instructions.

In the same way that the computer will interpret and execute the programmer's instructions, the contractor will interpret and direct execution of the architect's house plans and the dancer will interpret and execute the choreographer's notations. It may seem a bit farfetched to group together artistic endeavors such as writing, choreography, and music composition with technical endeavors such as electrical engineering, computer programming, and writing technical instructions. However there are claims that programming is an art (e.g., Molzberger, 1983) as well as analyses of writing and music composition in terms of their similarities to analytic problem solving (Flower & Hayes, 1980; Reitman, 1965).

Tasks become design tasks, having the feature that the product of the problem solving is a solution plan, for a variety of reasons. One reason to produce a complete solution plan prior to execution is because the solution plan is reusable. Thus a problem need not be solved step by step each time it is encountered. Computer programs, assembly and repair instructions, and standard house plans are examples of problem solutions for this purpose. A second purpose may be that the design itself conveys ideas in a pleasing manner. Choreography, story writing, and other creative design tasks have this purpose. A third and critical reason for designing is that many tasks are too complex to manage without planning and the cost of error too great. Early actions in problem solving constrain later actions in complex ways that cannot be anticipated, due in part to our limited cognitive capacity. In a puzzle problem (like cryptarithmetic), solution paths that deadend may be abandoned and alternate moves selected from a previous choice point. This trial and error method would be quite costly in building a house (Anderson, 1980). Instead, a solution plan, a design, is developed in advance of its

execution, and the task of the problem solver is to compose this solution.

Two fundamental activities in design task domains are composition and comprehension. Composition is the development of a design and comprehension results in an understanding of a design. The goal of this paper is empirically to investigate some aspects of design comprehension. However, difficulties in comprehension are intimately tied to difficulties in composition. For this reason, an argument is first developed that a critical feature of design composition, and a source of its difficulty, concerns multiple relationships between parts. This argument is then extended to explain sources of complexity in comprehension. New empirical work investigating comprehension processes in one design task domain, computer programming, is then presented.

COMPOSITION

Planning plays a central role in composition in design tasks. The same considerations of complexity that give rise to the design nature of a task make it unlikely that the designer will be able to compose the solution directly in the terms of the design language.[1] Thus, in all design task domains, there is a need to limit the complexity of the planning space. This can be accomplished by transforming the problem into a collection of simpler problems whose solutions can be comprehended. Most often this decomposition process must be repeated to reach a level where the solutions of subproblems can be specified at the level of detail required by the medium.

This orderly approach to planning in which a problem is decomposed into non-overlapping subproblems, which are themselves decomposed into subproblems, and so on, has been called step-wise refinement (Sacerdoti, 1977) and particularization (Reitman, 1965). In this method, each level of decomposition represents a more detailed plan for the solution. In many

design tasks, there is a "grammar" for the design that might well provide the basis for decomposition. For example, houses will contain a well defined configuration of rooms, literary and musical forms generally have a set of predictable parts. In many cases, the parts can be conceptualized as an orderly hierarchical structure. For example, given the task of writing a fugue (example from Reitman, 1965), the fugue can be viewed as exposition plus development plus conclusion. The exposition will contain thematic material plus countermaterial. Thematic material consists of motive plus development of motive, and so on. Similar decomposition rules have been developed for stories (e.g., Mandler & Johnson, 1977; Rumelhart, 1975).

There is some empirical support for design by stepwise refinement (Jeffries, Turner, Polson, & Atwood, 1981). However stepwise refinement does not provide a complete model of the composition process, a view that can be demonstrated both analytically (Reitman, 1965; Wilensky, 1983) and empirically (Hayes-Roth & Hayes-Roth, 1979). The main reason for this is that for any given decomposition that isolates subproblems, there are multiple relations between subproblems that require coordination; it is an idealization with no basis in reality that problems can be decomposed into non-overlapping subproblems. Any particular decomposition of a problem is based on a simplified model of the problem in which certain details are suppressed and others highlighted. For example, if the "grammar" of the design serves as the principle for decomposition, then certain structural components will be highlighted and other relationships "hidden." Consider for example a floor plan for a house, a decomposition that might be thought of as the blueprint. However, from this representation it is quite difficult to see what the exterior surfaces of the house will look like. It is also difficult to see a multitude of other relationships between rooms and floors, such as wiring and

plumbing systems. Thus, designing a house requires multiple decompositions to obtain a complete solution and this is reflected quite concretely in the size of the set of blueprints necessary to convey the many relations between components. Each blueprint is constructed to simplify the design space by highlighting some relations and ignoring others. Examples from other domains include circuit design, writing technical instructions, and computer programming. In electrical circuit analysis, one decomposition is made to facilitate the DC analysis and a second decomposition for the AC analysis (Shrobe, 1979). Writing technical instructions is difficult because instructions can be organized by goals (what is to be accomplished) or by temporal ordering (the sequence of steps that accomplishes the goals). When one organization is chosen, other relations must be inferred across subparts (Pennington, 1982). Computer program design is analogous to writing technical instructions and is developed extensively below as an example.

A characterization of composition that follows from this is as follows: A designer begins with a description of a problem that is quite abstract, for example to write a fugue, design a house, or write a computer program that achieves certain goals or follows certain constraints. The end product is a solution to the problem expressed in specific, detailed, executable actions or design elements. The transformation from initial problem description to final product must of necessity be one of successive refinements, that is, more and more constraints and details will be added to the solution plan as composition progresses. In most cases, successive refinement is aided by the use of multiple abstractions or decompositions in each of which some attributes of the problem are suppressed in order to concentrate on other attributes. It may be the case that a particular decomposition will serve as the "focal plan" with excursions into other planning spaces for some kinds of refinement.

Indeed, arguments about how to compose in technical design domains are often arguments about which abstraction is most useful and should serve as the organizing principle.

COMPREHENSION

Comprehension of solutions to design problems is as important as composition. In aesthetic domains, comprehension of art and literature leads to enriched enjoyment in proportion to depth of understanding. For designs that others will implement, such as house plans and technical instructions, comprehension is essential for their successful implementation. In other cases such as computer programming and writing, modifications made to the text require understanding of it.

In many ways, comprehension is the reverse of composition. Comprehension involves seeing the purpose of elements of the design or problem solution; it involves inferring goals by looking at plans. In following instructions, for example, "I don't understand" often means "I don't understand why I am instructed to do this" rather than "I don't understand what I am instructed to do."

If composition is characterized as successive refinement, then comprehension can be viewed as successive abstraction. The result of composition is a final problem solution plan, specified in primitive elements of the design medium. In the same way that the design is specified to preserve certain relationships among parts, these relationships must be detected for understanding. Like composition, a single form of abstraction may not suffice. To the extent that a design contains several forms of information, multiple abstractions are needed to recover all of them. For example, in electrical circuit analysis, AC and DC decompositions of the design will be made to fully understand the functioning of the circuit. To

recover some kinds of information about house design, it may be necessary to integrate information from multiple abstractions. Since the goal of the current paper is to explore psychological aspects of multiple representations by looking closely at experienced computer programmers' comprehension of computer program text, detailed examples from this domain are developed in the next section.

An important question for psychology concerns how effective mental representations are formed in the face of multiple potential abstractions that characterize the stimulus in overlapping ways. This is particularly important in areas of technical design in which arguments for and against particular composition methods are often based on claims concerning ease of comprehension and on claims concerning which organizing principles are central in mental representations.

In the next section of the paper multiple abstractions of computer programs are illustrated and related to current controversies in programming methodology and development. This framework is then used to view some related empirical and theoretical work on the topic. Finally, new empirical research is presented. Throughout I will argue by analogy that these concepts are relevant to the study of problem solving and comprehension in a wide range of tasks, and that the discussion is especially applicable to a subset of design tasks involving writing procedural instructions.

MULTIPLE ABSTRACTIONS IN COMPUTER PROGRAMMING

One feature of technical design in general and computer programming in particular is the development of artificial languages that sharply constrain interpretation, so that agreement on their semantics is universal. Even in this situation, in the use of programming languages created for human communication with machines, multiplicity occurs (Green, 1980; Green, Sime, &

Fitter, 1980; Pennington, 1982). This creates a problem for the skilled practitioner: Composition or comprehension of a problem solution may require multiple decompositions of a single system, but the programming language requires that the system be represented (in text form) by a single decomposition which has historically been related to the structural features of the computer system rather than to the cognitive requirements of the creator. This complexity has motivated the development of many programming aids although debates and research concerning their utility abound (e.g., Pennington, 1982; Shneiderman, 1980).

The composition task of the programmer is to transform a description of a problem in terms of what the program must do (goals) into a detailed list of instructions to the computer designating exactly how (procedures) to accomplish the goals. It is for this reason that programming can be viewed as writing procedural instructions; the programmer is designing a specific procedure for the computer to follow that accomplishes certain goals. Whereas composition in design tasks (in general) can be described as increasing refinement, for computer programming this means increasing proceduralization.[2]

For complex programming problems the composition of a program, i.e., the transformation from what to how, will require a series of intermediate representations, and their elaboration requires successive applications of strategies and knowledge from different domains (Pennington, 1982). Comprehension of a program may be viewed as the reverse series of transformations from how to what. That is, the person comprehending the program must put together various pieces to rebuild the meanings of the program. If indeed programming is the successive transformations of a description of a problem solution through the successive application of knowledge and strategies from different domains, then central questions

concern the nature of the representations and the nature of the programmer's knowledge, how it is organized and accessed, etc. If there were only one way to conceptualize a computer program then the nature of intermediate representations could be inferred on analytic grounds. However, there are multiple ways to conceptualize a program thus providing alternate abstractions that could serve as a basis for program plans.

An example of a very simple "toy" programming problem and four different abstractions of a solution to it are presented in Figures 1 through 5 (see Pennington & Grabowski, 1985 for an elaboration of this example). The problem (Figure 1) is to rearrange a table of codes so that all the codes of one type are moved to the first part of the table and all the codes of a second type are placed in the second part of the table, plus some other marking and printing requirements.

****
Insert Figure 1 about here
****

The first abstraction of the problem solution is structured in terms of the goals of the program, that is, what the program is supposed to accomplish or produce (Figure 2). It is labeled a "goal hierarchy" but could also be described as a decomposition according to the major program functions or outputs (cf. Adelson, 1984). The first level decomposition shows that the program will "do" or produce three things: a rearranged table, identifying labels for each code, and some printed output. At the next level, subgoals are specified for each higher level goal. For example, rearrangement of the table involves separating "A" codes from "B" codes, putting "A" codes first into the table, and putting "B" codes second into the table. Notice that in this representation there is no explicit information as to how these goals will be accomplished. For example, the "A" and "B" codes could be separated

and then counted up and then put back into the table. Alternatively, a single code from the table could be examined, classified as "A" or "B", added into the appropriate "A" or "B" counter, placed at the beginning or end of the table according to its "A" or "B" status, and then the next code would be examined. Some inferences about the ordering of events can be made from this representation on the basis of everyday knowledge; for example, a code must be classified before it can be counted as a member of the category, but details of the procedure to do this are not specified.

****
Insert Figure 2 about here
****

A second conceptualization is structured in terms of processes, operating on data objects, that transform the initial data objects into the outputs of the program (Figure 3). For example, Figure 3 shows that the data object "Table 1" is used by the process "Select 'A' Codes" to produce a list of "A" codes but "Table 1" is not itself transformed until it enters the process "Put 'A' Codes into Table 1" at which point it emerges from this process as a new version of "Table 1." Because the flow of each data object can be traced through the series of transformations in which it participates, this is called a "dataflow" representation. This representation is closely related to the goal hierarchy. For example the first level decomposition of goals is, in the goal hierarchy (Figure 2), to rearrange Table 1, Label Codes, and Print. These correspond to the final data objects at the bottom of the dataflow representation (Figure 3) which are Table 1, a Report, and Table 2. The goal hierarchy can be at least partly recovered from the dataflow representation by working up from the bottom of the dataflow representation although it requires the application of knowledge to infer the grouping of subgoals with their goals. However, in the dataflow representation, everything that happens to a

particular data object is readily available in a way that is not apparent from the goal hierarchy. In addition, this representation allows more inferences to be made about the order in which certain operations will occur than does the goal hierarchy representation. If an action (marked by a box, e.g., "Mark 'B' position with a 2" in the lower right of Figure 3) has a data object as an input (marked by an oval, e.g., "No. of 'B' codes") then the action cannot take place until the data object is available; thus the process that produces the data object (e.g., "Count 'B' codes") must execute prior to the process that consumes it ("Mark 'B' positions").

****
Insert Figure 3 about here
****

A third conceptualization is structured in terms of the sequence in which program actions will occur (Figure 4). This is called a control flow representation because the links betweeen program actions in this structure represent the passage of execution control, instead of the passage of data as in the dataflow representation. Traditional programming flowcharts are a standard expression of a control flow representation. This representation highlights sequencing information but conclusions about dataflow must be extracted by looking for repeated mentions of variable names. For example to find out in what events "No. of 'A' codes" participates (easily determined in the dataflow representation, Figure 3), it is first necessary to see that this quantity is represented by a variable called "Next 'A' Location Pointer" and then to track its use in the sequence of statements. To complicate things further, this variable is doing double duty as a counter of "A" codes and as a pointer to where the next "A" code goes in the table. This makes it difficult to extract goal information, even at a detailed level. So the sequence of statements involved in the subgoal "Count the number of 'A' codes" (Figure 2)

not only is embedded in statements serving an indexing function, but is also rather widely dispersed in the sequential representation (Figure 4).

****
Insert Figure 4 about here
****

A fourth conceptualization is structured in terms of the program actions that will result when a particular set of conditions is true (Figure 5). This representation is like a decision table in which each possible state of the world is associated with its consequences; it also resembles the production system condition-action pairs that are used in representing human procedural knowledge (e.g., Anderson, 1983; Newell & Simon, 1972). In this kind of representation, the program is viewed as being in a particular "state" at each moment in time, that is, some set of conditions exists. These conditions trigger an action, execution of the action results in a new state, the new state triggers another action, and so on. In this kind of representation, it is easy to find out what results if a given set of conditions occurs and also relatively easy to find out what set(s) of conditions can lead to a given action. This kind of "state" information is much harder to deduce from the other representations. However, information about the sequence in which actions occur and information about higher level goals are difficult to extract in the conditionalized action representation (Figure 5).

****
Insert Figure 5 about here
****

This analysis of the multiple points of view that characterize a computer program also applies to English language instructions, such as training manuals, recipes, knitting instructions, and assembly instructions, written for one's own future use or for another person's use. In these tasks, too, the writer wants to convey to another person what should be accomplished (goal

hierarchy), how to do it (sequential procedure), the sets of conditions under which particular actions should be taken (conditionalized action), and/or the set of transformations that a particular object should go through (dataflow). Part of the difficulty of writing clear instructions is due to the tradeoffs that inevitably occur in how much of each kind of information can be highlighted simultaneously. Uncertainty about the "best" way to write instructions may be largely due to uncertainty about which of these (or other) structures should serve as the organizing principle for the instructions.

Critical questions for comprehension research in computer programming and in other domains concern how these multiple abstractions are developed into effective mental representations necessary to comprehend a text or solve a problem. It is possible that one of these conceptualizations is more dominant psychologically due to features of the cognitive processing system and the organization of knowledge used in the comprehension process. For example, from computer programming research, Adelson (1984) suggests that a representation in terms of program goals (Figure 2) characterizes the "natural" or preferred representation of experienced programmers while a procedural representation (Figure 4) is most natural for novice programmers. It is also possible that the actual representational scheme used by the programmer will reflect task and programming language influences either in addition to or instead of the influences of knowledge organization. For example, different programming languages highlight different relations outlined in Figures 2-5 (Green, 1980; Green, et.al., 1980). Much has been written about how programmers think about programs but little of the work is empirical.

RELEVANT THEORY AND RESEARCH

Questions concerning expert skill, the nature of specialized knowledge used in a skill domain, and the nature of the mental representations used in

problem solving and comprehension have been addressed across a wide variety of skill tasks (see Greeno & Simon 1984 for a review). Some of the conclusions from this research are relevant to questions concerning effective mental representations in design tasks.

It is well known that experts in a skill domain are able to study a meaningful problem array and to remember more of it than are novices. However when presented with a scrambled arrangement of problem elements, experts and novices recall about the same amount of information. The result was first demonstrated in studies of chess masters and novices (Charness, 1976; Chase & Simon, 1973a, 1973b; de Groot, 1965) and has been replicated for a wide range of tasks, including not only other move games such as GO (Reitman, 1976) and bridge (Engle & Bukstel, 1978), but also a number of design tasks such as electronic circuit design (Egan & Schwartz, 1979), music composition (Halpern & Bower, 1982; Sloboda, 1976), and computer programming (McKeithen, Reitman, Reuter, & Hirtle, 1981; Shneiderman, 1976). The explanation for this empirical result has been that experts identify meaningful patterns in the problem array that are stored in memory as "chunks" of information. Thus experts recall more because their knowledge of the domain allows them to perceive and remember groups of information rather than individual pieces of information. The view is now widely held that superior organization of information is one key to expert performance.

In relation to the analysis of multiple representations in design tasks, these results suggest that there is indeed a dominant decomposition in comprehension such that for experts an abstract representation of the problem array is available quite quickly upon inspection of it. Much less is known about exactly which principles underlie experts' superior organization of information within domains and whether there are general principles across

domains that govern the nature of abstractions in comprehension. For the domain of computer programming, there has been some attention to the nature of program representation and the units or chunks that underlie its organization. This has been motivated in part by arguments over how programs ought to be structured, in part by the search for complexity measures to describe or evaluate program comprehensibility and also by the desire to extend understanding of skilled performance to an important complex task.

Almost all of the proposals for the nature of program chunks or cognitive units used in representing the meaning of a program refer to those units as "functional chunks." However the meaning of "function" is elusive in these various proposals. As one author put it, "Programmers understand programs by assimilating a group of statements that have a common function. This is a loose definition because it is not clear exactly what a common function is and how big a chunk can be" (Davis, 1984, p. 121). Nevertheless, the different proposals can be grouped loosely into two views: the Structured Programming (SP) view and the Plan Knowledge (PK) view.

According to the Structured Programming (SP) view, programs are conceptualized in terms of a limited number of control flow constructs. This view is most closely related to a control flow abstraction (Figure 4). While there are differences about the exact number and description of these control flow constructs, three basic building blocks are typically included: sequence in which control passes simply from one action to the next; iteration in which an action is repeated until a specified condition exists (commonly referred to as looping); and conditional in which control passes to different actions depending on which of two or more conditions is met (sometimes referred to as if-then-else).[3]

This view is called the Structured Programming view because of an

emphasis on disciplined control structuring by early SP advocates. Although there are other aspects to structured programming (and other methods of structuring besides control flow) advocates of SP hypothesized that programs organized according to this discipline are easier to understand and modify because they correspond to the programmer's mental organization.

One presentation of such a comprehension scheme proposes that a program, like other texts, has a micro-proposition text base connected by argument repetition (dataflow in programming terms) and a macro-propositional structure organized by control "primes" (prime meaning that the unit cannot be further decomposed into sequence, iteration, or conditional subunits) (Atwood & Ramsey, 1978; Curtis, Forman, Brooks, Soloway, & Ehrlich, 1984; Linger, Mills & Witt, 1979). In this view, program comprehension proceeds by identifying these sequence, iteration, and conditional units in the surface structure of the program and deriving their local purposes. These units then act as items that combine into higher order sequence, iteration, and conditional units, with higher level functions attached to units at this level. This process continues until the highest level is a single unit with an identifiable function. An example SP decomposition of a program segment written in COBOL (Figure 6.A) is shown in Figure 6.B, where "initialization sequence", "read loop" and "computation sequence" are the first level macrostructure control primes and higher levels of macrostructure are created by their combination. In this view dataflow connectivity between control prime units is a major source of processing difficulty. This corresponds to a view of programs as sets of instructions to be executed and suggests comprehension occurs in those terms.

****
Insert Figure 6 about here
****

As a psychological theory, the SP view is consistent with suggestions by other theorists (Atwood & Jeffries, 1980; Atwood & Ramsey, 1978; Davis, 1984; Mayer, 1979; Schneiderman, 1980) and has some indirect empirical support. One line of support is provided by evidence that programs have a kind of "phrase structure" in which the phrases are syntactically marked by keywords of the programming language: WHILE...DO is a marker that initiates loops; BEGIN goes with END to mark a sequence and so forth (McKeithen, et al., 1981; Norcio & Kerst, 1983). This mental structuring has been inferred by examining the free recall of programmers. In one study (McKeithen, et al., 1981) statements recalled most frequently by experts but not by novices corresponded to statements marking the phrase structure. In another study (Norcio & Kerst, 1983) higher proportions of correct to incorrect recall transitions (and vice versa) occurred at the boundaries of the hypothesized units, analogous to findings in sentence comprehension research that recall errors are greater over boundary transitions (Fodor, Bever, & Garrett, 1974; Mitchell & Green, 1978; Tejirian, 1968).

Additional support for this view is provided in a study of program comprehension in which programmers studied short programs composed of meaningful code, structured but meaningless code, or randomly arranged lines of code (Schmidt, 1983). More lines of both meaningful and structured but meaningless code were recalled compared to randomly arranged code. Further, longer study times occurred at control construct borders, in the same way that reading times are elevated at episode boundaries in story comprehension (Haberlandt, 1980; Mandler & Goodman, 1982).

Additional indirect support for this view was obtained in a study (Adelson, 1981) of subjective organization in programmers' (multitrial) free recall of randomly presented lines of program code. The randomly presented

lines contained statements that could be viewed as three routines (5 lines each) or as five syntactic groupings (3 lines each). Experts used program membership as an organizing principle and routines were grouped at a secnod level by procedural similarity, and not by the function of the routine.

In summary, the SP view suggests three main features of program comprehension: (1) Comprehension proceeds by segmenting statements at the detail level into phrase like groupings that then combine into higher order groupings. (2) Syntactic markings provide surface clues to the boundaries of these segments. (3) The segmentation reflects the control structure of the program. Thus in terms of the multiple abstractions of programs (Figure 2-5) sequence information should be readily available; dataflow connections that occur across unit boundaries should be relatively more difficult to infer; and function information should be least accessible since it is most closely related to dataflow and requires coordination across units.

According to the alternate Plan Knowledge (PK) view, programs are comprehended through the recognition of stereotypical functional units called plans (Rich, 1981; Soloway, Ehrlich & Black, 1983; Soloway, Ehrlich & Bonar, 1982). Plans correspond to a vocabulary of intermediate level programming concepts such as searching, summing, hashing, counting, etc., and there are hundreds (maybe thousands) of these rather than only a few. Like other forms of engineering and design, "there is a craft discipline among programmers consisting of a repertoire of standard methods of achieving certain types of goals" (Rich, 1980), and these standard plans determine comprehension and recall. A plan is a structure with roles for data objects, operations, tests, or other plans, and with constraints on what can fill the roles in a given instantiation as well as specifications as to dataflow and control flow connecting segments within plans. Plans accomplish things and are

hierarchically linked on the basis of function and role relations. That is, one plan may be used to accomplish the goals of a higher order plan (Soloway et al., 1983).

Plans are abstractions in the sense that there are usually multiple implementations of a given plan. Although data plans capture data abstractions and temporal plans represent control abstraction, plan representations of a program are based on dataflow relations. This is because much of the control structure in a program that is not mandated by dataflow requirements is arbitrary. There are also advocates of structuring programs by data structure (Jackson, 1980) and dataflow (Goldberg, Robson, & Ingalls, 1981) in part on the grounds that this better reflects the natural way of thinking about programs.

The PK theoretical position has been most fully elaborated by Rich, Shrobe, and Waters (Rich, 1980, 1981; Rich & Shrobe, 1979; Rich & Waters, 1981; Shrobe, 1979; Waters, 1979), who have developed a large set of plans based on their intuitions about programming, and also by Soloway and Ehrlich (Soloway & Ehrlich, 1984; Soloway, et al., 1983; Soloway, et al., 1982). In the PK view, understanding a program is to find a set of underlying plans so that parts of the program match the roles in the hypothesized plan. Comprehension of a program proceeds by partial pattern matches activating candidate plans, causing programmers to search for further evidence to instantiate a plan. According to this concept of comprehension the program is mentally represented as a set of linked descriptions, like blueprints, rather than as a set of instructions to be executed. In these descriptions function and dataflow relations are primary, not sequence or the detailed operations.

A PK decomposition of the COBOL program segment (Figure 6.A) in terms of underlying program plans is shown in Figure 6.C. This particular decompo-

sition is based on work by the MIT Programmer's Apprentice project (Rich & Shrobe, 1979; but see Soloway, et al., 1983 for a similar analysis). The segment consists of various plans including plans for reading through the inputs, counting, summing, and computing an average.

PK representations have also been invoked to explain how expert programmers chunk program text in recall tasks (Greeno & Simon, 1984) by arguing that plan knowledge is used to code the functions of the presented program. Details of the program need not be encoded because the programmer has only to expand a plan into one of its implementations to reconstruct the detail. This kind of claim corresponds to claims made in other domains that encoding efficiency is achieved on the basis of activating scripts (Schank & Abelson, 1977) or other kinds of content schemas such as goal/plan knowledge (Schank & Abelson, 1977; Wilensky, 1983).

For programming some indirect empirical evidence also supports the PK view. One kind of support for this position comes from evidence that dataflow relations are important in algorithm design (Kant & Newell, 1984) and program modification (Weiser, 1982) tasks. There is also evidence that experienced programmers are better than novices at inferring program function in a comprehension task (Adelson, 1984). These studies do not address questions about how functions and dataflow relations are inferred (i.e. by recognizing plans or some other way) but they do indicate the central role of program function in experts' understanding. Evidence more directly related to plans as critical elements in program comprehension is provided by Soloway and colleagues (Soloway & Ehrlich, 1984; Soloway, et al., 1982) using a cloze procedure to show that programmers will fill in a missing line of a program with a predicted plan element; that programmers have more difficulty comprehending a program in which the plan structure has been disrupted; and

that experts but not novices can resolve dilemmas when conflicting cues about which plan to instantiate are provided. In addition, Brooks (1975) has successfully simulated program composition by specifying a large set of program plans and processes that operate on them.

In summary, the PK view suggests the following features of program comprehension: (1) Comprehension proceeds by the recognition of patterns that implement known programming plans. (2) Plans are activated by partial pattern matches and confirming details are either sought or assumed. (3) The resulting segmentation reflects the dataflow structure of the program indexed by program function. Thus in terms of the multiple abstractions of programs (Figure 2-5), dataflow and function information should be readily available; sequence and detail operations should be less accessible.

The structures illustrated in Figures 6.B and 6.C represent potential alternative meaning structures in memory corresponding to the SP and PK views. There are several reasons to be interested in which of these views (SP, PK) better characterizes computer program comprehension. First, the two modes of comprehension have different consequences in terms of the kinds of information that are relatively easy or difficult to abstract from program text. This in turn is important in determining standards for computer programming practices, tools, languages, and education. More broadly however these two views of program comprehension mirror debates in other areas of text comprehension and composition. In general the debate centers on the ways in which different <u>kinds</u> of knowledge contribute to text understanding. One kind of knowledge that has been proposed to influence comprehension is abstract <u>knowledge of text structure</u>. For example, work on "story grammars" provides evidence that content free abstract knowledge about the type and form of components usually included in a story (setting, episodes consisting of

identifiable parts such as initiating events, goals, attempts, and consequences) provide organizing memory structures in story comprehension (e.g., Mandler, 1984; Rumelhart, 1975). A second kind of knowledge that has been proposed to influence comprehension is schematic content knowledge. For example, work on "scripts, goals, and plans" provides evidence that content specific knowledge about typical human action sequences and knowledge of typical plans that achieve certain goals provide organizing memory structures in story comprehension (Black & Bower, 1980; Bruce, 1980; Schank & Abelson, 1977; Thorndyke & Yekovich, 1980). Arguments concerning these claims in the domain of stories are difficult to resolve since both story texts and the plans involved in human action sequences tend to have the same or similar structures (Black & Wilensky, 1979; van Dijk & Kintsch, 1983). It can be seen that in programming using more traditional languages, text structure knowledge corresponds to structured programming units. That is, the units are few in number and abstract, a kind of "episode" of programs. Plan knowledge corresponds to schematic content knowledge and there are potentially thousands of such patterns. Plan knowledge also corresponds to the types of domain specific knowledge that has been implicated in various problem solving activities across a range of skilled problem solving tasks like chess (Chase & Simon, 1973).

The evidence cited above in support of each view of computer program comprehension, in terms of structured programming units (SP) or plan knowledge units (PK) is not definitive. For example superior recall of program statements introducing loops could reflect key statements that activate plan knowledge (McKeithen, et al., 1981). Evidence that experienced programmers have tacit knowledge regarding awkward program constructions does not necessarily imply that this knowledge leads to plan based mental representa-

tions (Soloway, et al., 1983). The research reported in the next sections provides a direct empirical test between these views of memory structures.

RESEARCH OVERVIEW

One effective technique for empirically investigating structures in memory is to index the relative distance between elements in a hypothesized structure by measuring priming effects in item recognition (McKoon & Ratcliff, 1980, 1984; Ratcliff & McKoon, 1978). In this method, subjects study one or more texts and are subsequently presented with a recognition test in which they must decide whether or not each item in the list was in the text they had just studied. An item of interest in the test list, a target item, is preceded in one condition by another item hypothesized to be in the same cognitive unit as the target and thus close in the memory structure. In a second condition the target item is preceded by an item hypothesized to be in a different cognitive unit and thus farther away in the memory structure. Under the assumption that activation of an item in the memory structure activates items close to it and especially those in the same cognitive unit, response time to the target preceded by an item in the same cognitive unit should be faster than response time to the same target preceded by an item not in the same cognitive unit (Anderson, 1983; McKoon & Ratcliff, 1980); that is, a priming effect should occur.

In the research reported in this paper, the priming technique was used to examine distances between program statements in program segments like the one shown in Figure 6.A. If the representations in memory of the meanings of the program segment correspond to the structures built by the SP or PK decompositions, then the relative amounts of priming between the items should be predicted by the relative distances between the concepts in the diagrammed

structures. For example, in Figure 6.B, the SP structure, statement 2 (MOVE 0 TO TOTAL-ORDERS.) should prime statement 4 (READ ORDER-FILE INTO ORDER-REC.) because they are in the same SP cognitive unit. However statement 2 (MOVE 0 TO TOTAL-ORDERS.) should not prime statement 13 (ADD ORDER-REC-QUANT TO TOTAL-ORDERS.) as much because they are not in the same SP cognitive unit. The PK structure (Figure 6.C) makes the opposite prediction: statement 2 (MOVE 0 TO TOTAL-ORDERS.) should prime statement 13 (ADD ORDER-REC-QUANT TO TOTAL-ORDERS.) because they are in the same PK cognitive unit and statement 2 (MOVE 0 TO TOTAL-ORDERS.) should not prime statement 4 (READ ORDER-FILE INTO ORDER-REC.) as much because they are not in the same PK cognitive unit. (Note that other co-varying features such as argument repetition and surface distance will need to be controlled by balancing these attributes across the set of items used.)

It is possible that neither of the theoretical decompositions (Figure 6) precisely describes the programmer's decomposition. In this case priming effects will not be obtained. However, failure to find priming effects is not informative as to what is wrong with the theoretical proposals. For this reason, additional measures of program comprehension are desirable. One additional measure is to ask programmers questions about their understanding of the program text in order to ascertain what aspects of meaning can be attained in limited time and to provide an assessment of learning relevant to interpretation of recognition memory test results.

Earlier we suggested that there are at least four kinds of relations between program statements that contribute to a complete understanding of the program: major functional relations specifying the goal structure of the program (Figure 2); dataflow relations specifying the sets of events in the program in which particular variables participate (Figure 3); control flow

relations specifying the execution sequence of statements (Figure 4); and state relations specifying the sets of conditions and resulting actions in the program (Figure 5). A fifth kind of information in the program consists of the detailed operations themselves, the actions corresponding to a single statement or less.[4]

The two general approaches to program comprehension that have been outlined (SP, PK) differ in terms of the kinds of relations between program elements that are treated as central in mental representations. Therefore, the two approaches lead to differing predictions about the kind of information that will be directly available to the programmer from the representation, or at a minimum, easier to infer from the mental representation. The SP view suggests that when a programmer studies a program, the meaning is built up from the bottom in terms of the operations binding together into the control flow units that are assigned local purpose. Major function and goal information is available only after these relations have been built. Thus the SP view stresses detailed operations, control flow, and then function in the representational hierarchy, suggesting that questions about detailed operations and execution sequence will be answered more easily (faster and with fewer errors) than will questions about major function and dataflow. The PK view suggests that when a programmer studies a program, function is inferred immediately when a programmer identifies a familiar stereotypic unit and the operations and data objects will be bound to the role slots in the hypothesized plan. The PK view stresses dataflow and functional dependencies as central in the representations, suggesting that function, dataflow, and detailed operation information should be available in that order. Neither view predicts that state information will be easy to extract from program text although some languages and applications (AI programming in LISP) stress these

relations.

Comprehension questions can be designed for a program text that ask specifically about these different kinds of relationships in the program. Examples of each kind of question for the program segment shown in Figure 6.A are: Will an average be computed? (major function); Is the last record in ORDER-FILE counted in COUNT-CLIENTS? (sequence); Will the value of COUNT-CLIENTS affect the value of ACTIVE-AVG? (dataflow); When ORDER-EXIT is reached, will ORDER-REC-ID have a particular known value? (state) is TOTAL-ORDERS initialized to zero? (detailed operation).

METHODS

Subjects

Professional programmers were used as subjects since they are truly experienced in the skill, compared to college students who have completed a few programming courses or even compared to computer science graduate students. Three years of professional programming experience was considered to be a minimum for participation in the research, corresponding to a criterion of experience that is sometimes used in recruiting programmer/analysts in industry. The usual contrast between novices and experts was not undertaken because novices cannot perform the tasks that experts routinely do in their work.

Subjects were selected from a pool of over 400 professional programmers who volunteered to participate in the research project in response to mail solicitations to Data Manager Association members, radio and television announcements, Chicago newspaper stories, and approaches to several Chicago-area businesses and research institutions. However, the choice of languages was constrained by the availability of experienced professionals for each language. If our sample of 400 is representative, then about 85% of

professional programmers program in COBOL, FORTRAN, and ASSEMBLER. The remaining 15% includes primarily PASCAL, PL/1, and BASIC programmers with a scattering of professionals using newer high level languages (BDL, IFPS). For this reason, subjects were drawn from the COBOL and FORTRAN programmers, defeating a goal of looking at conceptually distinct languages, but providing a basis for examining the generality of findings across languages.

A total of 80 professional programmers participated in this study, 40 COBOL programmers and 40 FORTRAN programmers.[5] Data describing the backgrounds of the subject sample are shown in Table 1. The average FORTRAN programmer was 37 years old at the time of the study, male (95% of the sample), majored in computer science or other science/engineering field, had completed some graduate level work beyond a bachelor's degree, knew 6 other programming languages, had taken 4 programming courses, had programmed for 14.5 years, had been a professional programmer for 10.8 years and had spent an estimated 12,306 professional programming hours on program coding, debugging and modification tasks. Forty-three percent of the FORTRAN programmers had taught at least one programming course. The average COBOL programmer was 35 years old, male (77.5% of the sample), had a college degree, majored in social science or humanities, knew 4 other programming languages, had taken 4 programming courses, had programmed for 10.5 years, had been a professional programmer for 9.5 years and had spent an estimated 11,196 professional programming hours on program coding, debugging, and modification tasks. Forty-five percent of the COBOL programmers had taught at least one programming course. As can be seen from Table 1, differences between FORTRAN and COBOL programmers in educational level, college major, number of programming languages known, and number of years programming (but not number of years as a professional programmer) were statistically reliable.

****

Insert Table 1 about here
****

Subjects were run over a period of eight months from July, 1983 to February, 1984. Each programmer was paid $10.00 for participation to cover transportation and parking costs.

Materials

The task devised to test between alternate comprehension schemes (SP, PK) was constructed over a series of pilot subjects with particular attention paid to constructing a task that professional programmers would consent to perform and would find somewhat interesting. The final task included the following components: programmers studied short segments of program text, answered comprehension questions about each segment, recalled from memory as much of the segment as possible and responded to a recognition memory test.

One of the program segments used in the experiment is shown in Figure 6.A. This particular segment is unusual because it is extremely simple. It was included because it has been used in many other studies of program comprehension and is frequently used as an example in articles (e.g., Soloway, et al., 1983). All other program segments developed for this research were drawn from four full length programs currently in use in Chicago-area computer installations. The four programs spanned a range of program types: an engineering application, a batch file update program, an interactive program, and a computational program. Two of these programs were originally written in COBOL and two in FORTRAN. Eight program segments were taken from the programs and modified slightly so that they met the following criteria: 1) Each comprised exactly 15 lines; 2) each accomplished something sensible in isolation (i.e. was comprehensible as a fragment that did something concrete); 3) each contained SP and PK units that differed in content.

For each of the 8 program segments, 6 comprehension questions were composed that varied according to the category of information about program relationships to which each pertained (see examples above). Also for each of the 8 segments, a recognition test list was composed so that priming effects between items could be examined. The critical recognition items (targets) were developed by finding in each segment, 4 groups of 3 statements each (4 triples) so that each triple consisted of two prime statements and a target statement that satisfied the following condition: One of the two prime statements had to be in the same hypothesized cognitive unit as the target statement according to SP theory but in a different hypothesized cognitive unit according to PK theory. For example, statements 4, 13, and 2 (Figure 6.A) form a triple. Statement 4 is the SP prime because statements 4 and 2 are in the same SP cognitive unit (Figure 6.B) but in different PK units (Figure 6.C). Statement 13 is the PK prime because statements 13 and 2 are in the same PK cognitive unit (Figure 6.C) but are in different SP units (Figure 6.B). Statement 2 is the target item since it appears in both pairs. Thus for each program segment, 4 target items were identified along with their SP and PK primes and the remaining 3 lines of code were designated as filler items. The targets were arbitrarily divided into two sets designated as "A" materials and "B" materials. It should be noted that there are other bases on which program statements might be associated in memory. For example, the surface distances between the prime and target statements differ for the example just given. In addition, some program statements have repeated arguments, others are very similar syntactically, and the direction (forward or backward) between the prime and target might differ within a triple. It was not possible to hold all of these other factors constant within any one triple, nor was it always possible to balance SP and PK primes on all

potentially confounding dimensions within each program segment group of four triples. However, the four potential influences mentioned above, surface distance, argument repetition, syntactic similarity and direction between prime and target were balanced over all 32 (8 segments, four targets per segment) SP-target and PK-target pairs.

Prime and target items were embedded in a recognition test list consisting of 22 items, 7 false items and 15 true items. The false items were written by using variable names that had occurred in the segment but were connected in the false item with an operation that had not connected them in the segment. False items were not "tricky" in the sense that they did not consist of misspellings or paraphrases of the program statements since the false items were not intended to be lures. There was no interest in analyses of errors beyond insuring that speed-accuracy relations would not interfere with interpretation of the results. One of the seven false items was a repeated item; the subject saw it twice in the test list. The fifteen true items consisted of 3 "filler" true items and 6 prime-target pairs. The six targets consisted of four targets presented for the first time and two of these targets were presented a second time. For the first time targets, primes were paired with targets so that one group of subjects saw PK primes immediately preceding A-targets in the recognition test list and SP primes immediately preceding B-targets. A second group of subjects saw SP primes immediately preceding A-targets and PK primes immediately preceding B-targets. For the repeated targets, the prime not seen before was paired with the target. Test lists were arranged in four different orders so that each prime-target pair appeared once in each quarter of the list, subject to the restrictions that a target item could not be placed in the first or second position of the test list and that primes had to immediately precede their

targets. All program segments and test list items were prepared in two programming languages, FORTRAN and COBOL.

Procedure

Subjects participated in a single experimental session lasting approximately 2.5 hours at the University of Chicago, Northwestern University, or their place of business. To begin the session an experimenter showed subjects the IBM Personal Computer to be used during the session and pointed out distinctive features of the keyboard. Thereafter, all instructions to the subjects were presented via the computer screen. The first part of the session consisted of general task instructions, detailed instructions concerning the use of editing features required during the task, and a practice trial.

General Task Instructions. Subjects were told that they would study a 15-line segment of code for three 90 second intervals; that between the study intervals they would be asked to respond to comprehension questions and would be given a memory task. They were instructed that their primary task was to come to a complete understanding of the code so that they could answer the comprehension questions accurately. We emphasized that their responses to the memory tasks should follow from their attempt to understand the code; that attempts should not be made to memorize the text or use special strategies for memorization. The exact task sequence was described and then demonstrated in the practice session.

Task Sequence. A 15-line segment of code was presented on the screen for exactly 90 seconds after which it disappeared and the afterimage was masked. Next an instruction appeared for subjects to position their index fingers on special keys marked YES and NO and to prepare to respond to two comprehension questions. Subjects initiated the first question with a keypress and then

responded to each question by pressing YES or NO. The second question was presented after a 500 msec delay. Response latencies and actual responses were recorded by the controlling program.

The next screen announced the free recall section and instructed subjects to start the free recall with a keypress. Subjects then typed in as much of the 15 line segment as they could recall. The line being typed was displayed at the bottom of the screen, with previously entered lines displayed on the top portion of the screen. Subjects were not constrained to enter recalled lines of text in presentation order as they could easily designate that a line be inserted in the already recalled material rather than having it added to the bottom of the text. Recalled lines could also be deleted or modified. Subjects had been instructed during practice to enter lines as they occurred to them and to enter whatever portion of a line occurred to them. The controlling program recorded each entry in the order entered, the pause time between entries, the typing time for each line, and a record of the final display.

This study-comprehension-recall sequence was repeated twice for a segment. This was followed by a third study-comprehension trial but this time it ended with a recognition memory test. Recognition started with a screen reminding subjects to position their fingers correctly, to respond YES or NO as quickly and accurately as possible, and not to pause during the list presentation. Subjects initiated the test with a keypress, with subsequent lines triggered by the previous response. A blinking arrow appeared for 250 msec between each line presentation. The response and the response latency for each item were recorded.

The above sequence was repeated for each of the eight 15 line segments. Subjects took a break between the fourth and fifth segments. At the

conclusion of the session, subjects filled out a detailed background questionnaire and responded to questions posed by the experimenters about their own reactions to the experiment and their own programming work. Their responses to the questions and their general observations were tape recorded.

These procedures were established by extensive pilot testing. For example, the study time of 90 seconds was chosen to insure high levels of recognition accuracy and adequate levels of segment comprehension. The comprehension questions were inserted before the recall and recognition tasks to focus subjects on the comprehension aspects of the task rather than on the memory requirements and to discourage inclinations to rehearse or retain a visual image of the text. Initially the program text was not formatted or indented but this elicited complaints from pilot subjects so stimulus segments in this study were indented according to widely accepted conventions.

Design

The program segment and test list materials were used to form the basic research design: 2 (languages) × 4 (orders) × 2 (subject groups) × 2 (SP, PK prime types) × 2 (A, B sets of target items) with language and order as a between subjects factor and with subject groups, prime type, and materials set forming a 2×2×2 repeated measures latin square. In this design, comparisons between target response times for different prime types will be a between-subject comparison. A rearrangement of this design using first and second presentations of only those target items that were repeated in the test lists allows a within subject comparison between target response times for different prime types. This is a comparison of secondary interest because repetitions of true items in the test list are potentially confusing and may add variability to response times for these items.

Using this design, we sought evidence as to whether programmers' mental

representations of program text reflected structural distances hypothesized by the SP view, the PK view, or neither view. Specifically, support for the SP view is obtained if response times to targets preceded by an SP prime are reliably faster than the same targets preceded by a PK prime. If this is the case, we can infer that the items specified by the SP theory as forming a cognitive unit are in fact "closer" in memory than are the items specified by the PK theory. Alternatively, support for the PK view is obtained if response times to targets preceded by a PK prime are reliably faster than the same targets preceded by an SP prime. Finally, if some response times to PK-primed targets are faster and other response times to SP-primed targets are faster, then no inferences may be drawn regarding which of the formulations more accurately portrays the nature of mental representations.

Response times and error rates for different kinds of comprehension questions provide an additional measure regarding relations that dominate in mental representations. Specifically, if support for the PK theory is obtained with the recognition response times, then we expect to see fewer errors and faster response times for function and dataflow comprehension questions. Alternatively, if support for the SP theory is obtained with the recognition response times, then we expect to see fewer errors and faster response times for detailed operations and control flow comprehension questions.

RESULTS

Recognition Memory Data.

Recognition Memory Accuracy. The recognition test list for each of the 8 segments consisted of 22 recognition items for a total of 176 items. Approximately one-third of the items (7 per list) were NO items (false) and the remaining two-thirds were YES items (true). Of the seven false items, one

was a repeat of a previous false item. Of the 15 true items per list, three were fillers, six were primes that differed in content between the experimental conditions, four were target items seen for the first time in the list, and two were target items repeated in the list.

Overall recognition accuracy, measured by percent correct, averaged 92.1% with a range from 73.9% to 99.4%. FORTRAN subjects made fewer recognition errors (6%) compared to COBOL subjects (9.8%), $F(1,64) = 7.36$, $p < .01$. Recognition accuracy did not differ for subjects assigned to the two experimental conditions, nor were there any differences in recognition accuracy due to order of presentation of the program segments (Fs less than 1). Three subjects had accuracy scores less than 50; two of these had comprehension scores near chance levels as well.

Comparisons between percent correct for true (YES) items and false (NO) items revealed that error rates for these two types of items were equivalent for FORTRAN subjects (93.7% correct for true items, 94% correct for false items) but not for COBOL subjects (91.3% correct for true items, 87.6% correct for false items), $F(1,64) = 8.33$, $p < .01$ (interaction between language and repeated measure of true/false items). Because of this, overall recognition accuracy may be better indexed by taking the difference between the hit rate (% correct for true items) and the false alarm rate (1 - % correct for false items), since this adjusts overall accuracy for guessing and better reflects a subject's ability to discriminate between true and false items. (For example a subject responding true to all items would have 67% correct but the difference between hit rate and false alarm rate would be 0. Similarly a person performing at a chance level would have 50% correct and a difference score of 0.) Recognition accuracy, measured as the difference between hit and false alarm rates, averaged 83.2 over subjects with a range from 38.3 to

99.0. Using this measure, FORTRAN subjects still had higher levels of recognition accuracy than did COBOL subjects (87.6 vs. 78.9; $F(1,64) = 8.38$, $p < .006$) and again, differences were not found between different experimental conditions or among different orders of segment presentation.

We are primarily concerned with the analysis of the target recognition items (four per segment test list, divided into 2 A-targets and 2 B-targets). Error rates across subjects and across both sets of target items averaged 92.6% correct. Error rates differed by language (FORTRAN 94.2%, COBOL 90.9%; $F(1,64) = 4.19$, $p < .05$) but did not differ by experimental condition or by material set (A-targets, B-targets) Fs less than 1). Thus, in examining comparisons between response times for target items, a constant error rate of 8-10% may be assumed.

Overall Recognition Response Times. Response times for each recognition test item were measured from the time the recognition test item appeared until the YES/NO response keypress. The test items (single lines of program text) varied in length (mean 6.0 syllables, range 2 to 16 syllables) so response times of different items cannot be compared directly. Since the experimental comparisons of interest are between response times to identical items, no attempt was made to correct response times for reading time. However, to gain an idea of the overall response patterns we will look at average response times for different types of items, keeping in mind the variability in item length.

Over all items, response times ranged from .540 seconds to 28.230 seconds, with an average response time of 2.752 seconds. Correct responses averaged 2.670 seconds compared to incorrect responses, which averaged 3.667 seconds, $F(1,76) = 63.60$, $p < .000$. This suggests a difficulty relationship between speed and accuracy rather than a speed/accuracy tradeoff. All

incorrect response times were discarded from further consideration as were responses to all false (NO) items.

Considering only correct YES items, the average response time was 2.670 seconds. Although variability in response times was large, times such as the 28.230 seconds cited above constitute extreme deviations and a rule was formulated to identify extreme response times in the data so that they could be excluded from final analyses. A response time for a given subject and item was labeled extreme if it was more than 2.5 standard deviations from the subject's mean response time over correct responses, and if it was simultaneously greater than 2.5 standard deviations from the mean response time for that particular item computed over subjects in the same subject group. In addition, all response times greater than 10.0 seconds were considered extreme. About 1.9% of the response times were identified as extreme and their removal lowered the average response time by about 150 milliseconds and reduced variability. For example the "cleaned" average response time for correct YES items was 2.512 seconds compared to the previously mentioned 2.670 mean over these items. All analyses were performed on cleaned and uncleaned data and in no case was the direction of differences between means altered by the removal of extreme response times. All further results are reported using cleaned data.

For each of the 8 segments, 3 of the true test list items were designated as filler true items and were not involved in any experimental manipulation. Analyses of response times on these items reveal differences between language and subject groups that cannot be attributed to experimental treatments. Subject groups within language differed from each other, with FORTRAN Group 2 responding an average of 510 milliseconds faster than FORTRAN Group 1 and COBOL Group 1 responding an average of 272 milliseconds faster than COBOL

Group 2, $F(1,76) = 7.95$, $p < .007$ with subjects as random factor, $F(1, 46) = 39.698$, $p < .000$ with materials as random factor. Therefore, in examining response times for the target items of interest, we expected to have to correct for the group of the subject within languages.

The items of interest for further analyses are those designated as target items because each was preceded (primed) by an item hypothesized to speed up the response time to the target item, due to a relationship between the items in memory. If the priming item was answered incorrectly, then we cannot be assured that the item was in memory. Therefore, responses to targets preceded by an incorrect response to the priming item for that target were excluded from the analyses. Thus target response times include only those correct YES responses preceded by a correct YES response. This represents about 83% of the original 2,560 target responses (first presentation), or 2,109 observations available for analysis.

In addition to the 32 target items, each subject saw 16 of these targets repeated, thus all targets were seen a second time by one-half of the subjects. Times for these repeated targets were computed as a difference score between the first presentation and the second presentation. The average difference score for repeated targets was +133 milliseconds indicating that on average subjects responded true to the second presentation of a target 133 milliseconds faster than to the first presentation. Of course, for both the targets and repeat targets, we are interested in the effects of the prime item on the target response time.

Effects of Primes on Target Response Times. Each subject saw 4 target statements from each of the 8 program segments for a total of 32 target response times. Half of these target items were designated as material set A items and half as material set B items with 16 items in each set. After

removing extreme response times and response times of items for which a prime error was made (see previous section) a mean was computed for each subject for each of the two target sets (A,B). These means were used in a 2×2×4×2 repeated measures analysis of variance with language (FORTRAN, COBOL), subject group (1, 2), and test list order (4 orders) as between subjects factors and type of prime (PK, SP) as a within subject repeated measure. The subject group by prime type interaction corresponds to a within subject factor of material set (A, B), and subjects are an implicit random factor in this analysis, hereafter called the "subjects" analysis. A mean was also computed for each item for each subject group. A second 2×2×2 analysis of variance was performed on this data, treating materials as a random factor, with language (FORTRAN, COBOL), and material set (A, B) as between items factors, and prime type (PK, SP) as a within item repeated measure. In this case the materials by prime type interaction corresponds to a within item factor of subject group (1, 2). This analysis is referred to as the "items" analysis.

The question of primary interest is whether target response times are faster (1) when the target is immediately preceded by an item from an hypothesized SP cognitive unit, or (2) when the target is immediately preceded by an item from an hypothesized PK cognitive unit, or (3) there would be no difference between priming conditions. Examination of the cell means reveals that there are multiple influences on target response times (Table 2).

****
Insert Table 2 about here
****

As predicted by a structured programming (SP) analysis of program comprehension, responses to SP-primed targets are on average 105 milliseconds faster than responses to PK-primed targets, $F(1,64) = 4.51$, $p < .04$ (subjects analysis, Table 2.A), $F(1,60) = 3.72$, $p < .06$ (items analysis). Considering

only subjects whose comprehension scores were in the top quartile (assuming that these subjects had a more complete understanding of the program segments), we see (Table 2.B) that the SP-primed target speedup is larger, 237 milliseconds, $F(1,15) = 8.35$, $p < .02$ (subjects analysis), $F(1,59) = 3.60$, $p < .06$ (items analysis). Comparisons using the within subject repeated target data show the same advantage for SP-primed targets although the effect for repeated targets is statistically unreliable due to increased variance in repeat target times.

Target response times also differed for the A and B material sets, for the two languages, and for the subject groups within each language. Responses to B materials took an average of 261 milliseconds longer than responses to A materials, $F(1,64) = 28.18$, $p < .000$ (subjects analysis), $F(1,60) = 2.32$, $p < 0.14$ (items analysis), and this difference was larger for COBOL items than for FORTRAN items, $F(1,64) = 6.03$, $p < .02$ (subjects analysis, not significant for items analysis). COBOL subjects took longer to respond in general, $F(1,64) = 4.91$, $p < .04$ (subjects analysis), $F(1,60) = 4.88$, $p < .04$ (items analysis) and the same subject within language differences that were apparent in response times to filler trues hold for targets as well, $F(1,64) = 5.29$, $p < 0.03$ (subjects analysis), $F(1,60) = 34.54$, $p < .000$ (items analysis). The array of effects can be seen more easily graphically (Figure 7). In Figure 7.A means are adjusted to take out the effect of subject group showing the SP priming effect for both languages, the effect for materials set and the slight interaction between materials and language. In Figure 7.B, means are adjusted for the effect of materials sets, again showing the SP priming effect for both languages and the effect for subject group within language.

It is not surprising that materials and subject differences account for a major portion of variability in time to recognize program statements.

However, on top of these differences, program statements are consistently recognized faster when immediately preceded by a program statement in the same control flow unit (SP theory). This result strongly supports the mental organization of program text proposed by a structured programming analysis. On the basis of this result, we are led to expect certain converging results in the programmer's responses to comprehension questions about these same program statements.

****
Insert Figure 7 about here
****

Comprehension Data

Two comprehension questions that could be answered YES or NO were posed after each of the three study trials for each of the eight program text segments resulting in 48 items total. The 48 comprehension questions required that the programmer make inferences about five different kinds of relationships in the program text, or, to retrieve this information directly if the inferences had already been made and stored as part of the programmer's mental representation. The recognition memory results discussed above lead us to expect that questions concerning detailed program operations and control flow relations will be most accessible to the programmer as predicted by the SP theoretical analysis. We no longer expect to see advantages for dataflow and function relations as predicted by the PK theoretical analysis.

Overall Comprehension Accuracy. The correct response for 26 of the 48 comprehension items was YES, thus chance responding could yield a comprehension error rate of 46% to 50%. Subjects' comprehension error rates (percent incorrect) ranged from 6.2% to 52.1% with an average of 26.0%. In considering error rates for comprehension questions, it should be remembered that programmers were answering questions after very limited study time. One-

third of the questions occurred after only 90 seconds of study; one-third after 180 seconds; and one-third after 270 seconds. An average of 74% correct under these conditions reflects the high skill level of the programmers. Novice programmers are not able to perform this task at higher than chance levels of comprehension. Four experienced subjects (2 FORTRAN, 2 COBOL) performed near chance levels (> 40% errors) with one of the four performing below chance (> 50% errors), however these data were included in all analyses.

On average, FORTRAN subjects made fewer comprehension errors than COBOL subjects (24.0% vs. 29.0%), $F(1,64) = 6.70$, $p < .02$. Comprehension accuracy did not differ for subjects assigned to the two experimental conditions, nor were there differences due to the presentation order of the program segments (Fs less than 2).

Most of the programmer background variables (Table 1) are unrelated to comprehension performance on our task. For example, comprehension accuracy levels were equivalent for males and females, $F(1,78) = 1.32$; for different educational levels, $F(3,76) = 0.64$; and for different college majors, $F(3,71) = 1.16$). This is important because it suggests that any differences in performance between language groups will not be accounted for by sample differences in sex, education, and college major (see Table 1). There were also no differences in comprehension accuracy between programmers who had taught programming and those who had not, $F(1,78) = 0.96$.

Correlations were computed between the remaining (continuous) background variables and comprehension performance scores (percent correct). Variables showing a statistically reliable relationship ($p < .05$) with comprehension were the number of programming languages known by the programmer ($r = +.24$), number of years as a programmer ($r = +.22$), age of the programmer ($r = -.15$), and the number of programming courses taken ($r = -.14$). Variance accounted for

by number of languages known is completely redundant with variance accounted for by the other three variables and a multiple linear regression expressing comprehension as a function of age, years programming, and number of courses accounts for 25% of the variance in comprehension performance. Age and years programming are highly correlated (r= .75) and each accounts for no more than 5% of the variance in comprehension when considered alone. Together, the two variables account for 23% of the variance suggesting that when experience is controlled, the younger programmers did better and conversely, when age is controlled, the more experienced programmers did better on the comprehension task. Although only accounting for 2% of the variance, programmers who had had fewer courses did slightly better on our task.

Background profiles (using variables shown in Table 1) were computed for programmers in the top and bottom quartiles according to comprehension score. Top comprehenders (mean 15.2% errors) differed reliably from bottom comprehenders (mean 37.0% errors) on only two variables: years programming (14.6 years for upper quartile vs. 10.3 years for lower quartile), $F(1,33) = 4.45$, $p < .05$; and number of programming courses taken (2.6 for top vs. 4.4 for bottom), $F(1,33) = 4.90$, $p < .04$.

Comprehension Accuracy by Item Information Categories. Our main interest in comprehension accuracy is in differences that might occur between items in different information categories, that is, between questions asking about different kinds of information in the program text. Referring to the earlier conceptual analyses (Figures 1 through 5), the 48 comprehension questions comprised 10 questions about detailed program operations (operations questions), 9 questions about program execution sequence (control flow questions), 9 questions about program dataflow (dataflow questions), 10 questions about program condition-action relations (state questions), and 10

questions about major program functions (function questions). An assumption is made that higher error rates for questions in a particular information category imply that the information in that category is less easily accessed or computed from the memory representation. Under this assumption predictions derived from Structured Programming (SP) theory (Figure 6.B) are that operations and control flow questions will be most easily answered and that dataflow and major functions will be more difficult to infer. Predictions derived from Plan Knowledge (PK) theory (Figure 6.C) are that dataflow and major function information will be most accessible with operations and control flow less accessible.[6] State information is not accessible under either formulation.

Error rates were computed for each subject for items in each information category (percent of items missed in the category) and for each item (percent of subjects missing an item). The five scores per subject were submitted to a language (FORTRAN, COBOL) by information category (operations, control flow, dataflow, state, function) repeated measures analysis of variance (with subjects as random factor) and the item scores were submitted to a language by information category analysis of variance (with items as random factor).

Information category of the comprehension question affected error rates, $F(4,312) = 26.75$, $p < .000$ (subjects analysis, mean standard error = 2%), $F(4,86) = 2.87$, $p < .03$ (items analysis, mean standard error = 4%). Examination of the mean error rates (Table 3.A, third column) reveals that the ordering of difficulty of the information category questions is that predicted by the Structured Programming (SP) analysis: Questions about detailed operations and control flow relations are answered most accurately (15%, 21% errors respectively) while more errors are made on dataflow, state, and function items (28%, 30%, 34% errors, respectively).

****
Insert Table 3 about here
****

When means are equally weighted for information category, overall comprehension levels for FORTRAN and COBOL programmers still differ reliably when subjects are the unit of analysis, $F(1,78) = 6.01$, although with items as the unit of analysis, variability among items swamps this difference, $F(1,86) = 1.10$. In addition the pattern of error rates for information categories may differ for the two languages (Table 3.A, first and second columns), $F(4,312) = 8.72$, $p < .000$ (subjects analysis), $F(4,86) = .827$ (not significant, items analysis). This pattern is displayed graphically in Figure 8.

****
Insert Figure 8 about here
****

The comprehension pattern across information categories for FORTRAN subjects (Figure 8) is: Questions about operations and control flow are answered most accurately; questions concerning major function are answered next most accurately, followed by dataflow and state questions. This yields an inverted U shaped function with operation, sequence and function depressed, showing lower error rates. The most noticeable difference between the FORTRAN and COBOL patterns is the elevated error rate on major function questions for COBOL subjects, creating an increasing linear function across information categories. In addition to the elevation of major function question errors, control flow questions show lower accuracy and dataflow questions show slightly higher accuracy.

The difference in patterns could be due to one or both of two factors. First, the languages themselves could yield differences in ease of comprehension such that control flow and function information are "easier" to extract from FORTRAN than from COBOL text, and dataflow information is "easier" to

extract from COBOL than from FORTRAN. Alternatively, the pattern of differences might be due to lower comprehension levels for the COBOL subjects and thus the COBOL pattern could reflect an earlier stage in the comprehension process and the FORTRAN pattern a more advanced stage. Under the latter interpretation, FORTRAN subjects, having understood more of the operation and control flow information may have progressed further and could therefore either compute or retrieve function information from memory. COBOL subjects, it would be argued had not yet reached this stage in the comprehension process after the allotted study time.

In order to examine these explanations, subjects were divided into quartiles (within language) on the basis of their overall comprehension accuracy scores. If the overall FORTRAN pattern represents a more advanced stage of comprehension, then the upper quartile COBOL subjects should show a pattern more like the FORTRAN aggregate and the lower quartile FORTRAN subjects should show a pattern more like the aggregate COBOL subjects. Alternatively, if the differences in aggregate patterns reflect features of the language, then those differences will not be altered in the patterns for top and bottom quartile comprehension subjects. The comprehension question means for top and bottom quartile subjects are displayed in Tables 3.B and 3.C and the pattern is shown graphically in Figure 9.

****
Insert Figure 9 about here
****

Consistent with the "stage of comprehension" explanation, bottom quartile subjects show the COBOL aggregate pattern with major function question error rates elevated while top quartile subjects show the FORTRAN aggregate inverted U shaped pattern. A quartile by information category interaction, the statistical manifestation of this pattern, is only marginally significant,

$F(4,64) = 2.43$, $p < .06$ (subjects analysis), $F(4,86) = 2.02$, $p < .10$ (items analysis).

Some language specific features are also retained by the information category patterns, namely lower error rates for control flow questions for FORTRAN subjects (both quartile groups) and lower error rates for dataflow questions for COBOL subjects (both quartile groups). This is consistent with an explanation attributing differences to language specific features that affect the ease of extracting information from the text. However, as in the analysis of the complete subject sample, this interaction between language and information category shows statistical reliability only in the subjects analysis, $F(4,64) = 3.99$, $p < .006$, not in the items analysis, $F(4,86) = .619$.

In summary, after limited study time, experienced programmers' comprehension errors were strongly related to the kinds of inferences required to respond to the question, and to the language in which the programs were written (Table 3). Questions about program operations and control flow relations in the programs were answered correctly more often than questions about dataflow relations and program states. Errors were made most frequently when inferences about program function were required. This general pattern is most consistent with the SP theoretical formulation (Figure 6.B). Differences between FORTRAN and COBOL programmers and between top and bottom quartile comprehenders support two further conclusions. Inferences about program function (what it does) are the most difficult and appear late in the comprehension process. The ability to make function inferences after limited study time is the most discriminating difference between programmers displaying different levels of comprehension in our task. Furthermore, there may be language specific features that affect the ease of certain kinds of

inferences; FORTRAN programmers were consistently better on inferences about control flow while COBOL programmers were more accurate in responding to questions about program dataflow relations.

Comprehension Question Response Times. Response times to comprehension questions provide additional information about the relative difficulties of different kinds of inferences in comprehension. After correcting for reading time, it is assumed that a relatively faster response time to a question implies less processing has gone into constructing a response to the question. Faster responses could be due to direct retrieval of the information, to assessment of plausibility given retrieval of higher level information, or to fast computation of the response given retrieval of related information. Slower responses imply extensive search or burdensome computation from retrieved information (Reder, 1982; Glucksberg & McCloskey, 1981). For example, if major function information is stored directly and provides the macrostructure (Kintsch & van Dijk, 1978) of the text representation (Adelson, 1984; Atwood & Ramsey, 1978; Brooks, 1983; Shneiderman, 1980), then the fastest response times are predicted for major function questions, with slower responses predicted for more detailed comprehension questions like those in the operations category. However, if frequent errors correspond to the need to assemble responses at the time of questioning then the error rate data above lead us to expect that responses to operations questions will be faster than responses to questions about program function.

First, it is necessary to examine some attributes of the response time measure. Subjects took an average of 7.62 seconds to respond to a comprehension question, taking longer on average to respond incorrectly (mean 8.42 seconds) than to respond correctly (mean 6.80 seconds), $F(1,78) = 21.38$, $p < .000$; thus errors were due primarily to the difficulty of the question for

the subjects rather than due to responding too quickly. All further analyses were restricted to response times for comprehension questions answered correctly.

Response times for questions answered correctly (mean 6.80 seconds) were "cleaned" by removing all times that were more than two and one-half standard deviations from both a subject's mean response time over items and an item's mean response time over subjects (mean 6.22 seconds for cleaned response times). This insures that mean differences are not exaggerated or distorted by the influence of outliers although removing outliers in the following analyses did not affect any of the results obtained. All means reported represented cleaned response times.

Lengths of the comprehension questions varied (from 8 to 19 syllables, mean = 13.2 syllables) so response times need to be adjusted for reading time in order to compare response times between different information category question sets. Correlations between response times and numbers of syllables in the comprehension question ranged from -.06 to +.58 and averaged .22. Thus variability in response time is not strongly related to reading time and is not the same for all subjects. Therefore, assumptions underlying use of a response time per syllable measure are not met in this task, a sensible conclusion given the relatively long response times. Use of a residual response time after correcting subjects individually for number of syllables appears to be most suitable.[7]

****
Insert Table 4 about here
****

Of primary interest are the response times by information category of the comprehension question (Table 4) for both raw response times and residual response times. For all subjects, comprehension questions about detailed

operations, major functions, and dataflow were answered faster than questions about sequence, and questions about program state were slowest (Table 4.B, COMBINED column), $F(4,312) = 48.61$, $p < .000$ (subjects analysis, raw response time measure), $F(4,312) = 31.13$, $p < .000$ (subjects analysis, residual response time measure), $F(4,86) = 4.28$, $p < .004$ (items analysis). This pattern of response times for questions answered correctly closely resembles the pattern of errors for upper quartile subjects (Table 3.B, COMBINED column), with the exception that sequence question response times are relatively slower than the percent errors data would lead us to expect. Even this discrepancy disappears when examining response times to questions with a correct YES response (Table 4.B, YES QUESTIONS COMBINED column). A YES comprehension question asked about an operation, sequence relation, function, or state that was true of the program while a NO comprehension question asked about a relationship that was not true of the program. In general, YES questions were answered more quickly than were NO questions (-.11 versus .00 over all questions) but this difference was greatly exaggerated for sequence questions (-.16 versus .48). If we consider the YES questions to be more directly related to the retrieval of information from memory, then the pattern of response times directly reflects the pattern of comprehension question errors. That is, operations and sequence questions were anwered (YES) most quickly, followed by dataflow and function questions, followed by state questions (Table 4.B, YES QUESTIONS COMBINED colum).

Mean response times did not differ overall between languages (Fs less than 2) but the pattern of response times for information categories differs by language, $F(4,312) = 5.13$, $p < .001$ (subjects analysis, raw response times), $F(4,312) = 2.92$, $p < .03$ (subject analysis, residual response times). The differences in information category difficulty for the two

languages as indexed by comprehension question response times (Figure 10) again resemble upper quartile subjects' percent error data (Figure 9). COBOL subjects take longer to respond to function questions than do FORTRAN subjects, as expected from the percent error data. However, FORTRAN subjects' control flow inferences show relatively longer than expected response times. This latter discrepancy again disappears when considering only YES sequence questions for FORTRAN subjects.

****
Insert Figure 10 about here
****

Thus, neither the comprehension question response time data nor the percent error data support a view of comprehension in which major function information serves as an organizing structure in memory. The pattern of results support a different view: (1) Detailed operation information is stored directly, organized by control flow units (lowest percent errors, fastest response times). (2) Control flow inferences are thus easy to retrieve or compute (low percent errors, low moderate response times). (3) Some dataflow and major function information is readily available (moderate response times) although when not stored, it is not easily computed (higher percent errors). (4) Program state inferences are difficult to compute (long response times, high percent errors).

CONCLUSIONS

The research results provide evidence that the dominant memory representation, formed during comprehension of short program texts, is organized by a small set of abstract program units related to the contol structure of the program. More specifically, of the four program abstractions presented earlier (Figures 2 through 5), relations captured by the procedural, control flow abstraction (Figure 4) appear to be central in comprehension in

our experimental task. Futher, the nature of the mental unitization of these relations corresponds to the basic program building blocks of sequence, repetition, and selection identified by early advocates of Structured Programming.

Both recognition memory results and comprehension question results in the current study converge to support this conclusion. In the recognition memory test, recognition occured faster when a statement was immediately preceded by a statement in the same structured programming unit than when it was immediately preceded by a statement that was not in the same structured programming unit. Thus, statements in the structured programming unit showed a priming effect on recognition for other statements in the same unit. This implies that statements in the same SP unit were closer together in programmers' memory meaning structures. This priming effect cannot be accounted for by the text surface distance between the statements, by syntactic similarity between statements, or by argument repetition since these features were controlled by counter-balancing test items.[8] Futhermore, responses to comprehension questions about control flow relations and program operations were answered faster and with fewer errors than were questions about dataflow and function relations, supporting the idea that control flow and operation information is easier to access in memory.

Examination of data from the programmers with the highest comprehension scores strengthens these conclusions and leads to a further speculation that segmentation on the basis of control flow relations occurs prior to comprehension of major program functions and dataflow relations. First, the priming effect for SP cognitive units was strongest for top comprehenders. Second, top comprehenders' fast and error free responses to detail and control flow comprehension questions were accompanied by a disproportionate decrease

in response errors to function questions. While one might argue on the basis of the error data alone that there is a priority for function information in top comprehenders' memory representations, the priming results and response time data do not support such a conclusion.

These empirical results suggest a view of program comprehension in which the meaning of program text is developed largely from "the bottom up." That is, the text is first segmented according to simple control patterns segregating sequences, loops, and conditional patterns. At this level some specific inferences are made concerning the procedural role of the segment. For example, a sequence at the beginning in which zeros are assigned to variables may be designated "initialization of variables" (see Figure 6.B), without regard for the role of those variables in later computation. Another sequence may be designated as "something is calculated." A loop repeats whatever sequence is contained within it. A conditional pattern directs control to alternate sequences.

Dataflow and function connections often require integration of operations across separate segments. For example, calculation of an average involves an initialization, a running sum, and final calculation. As in Figure 6.B, these occur in three separate procedural units. The research results suggest that these connections are made later in comprehension, and a distinguishing feature of the programmers with the lowest comprehension scores is that they are not made correctly or at all.

These conclusions are consistent with previous empirical results implicating structured programming or procedural units in studies of programmers' memory for program text. For example, findings that statements syntactically marking sequence, loop, and conditional program units are most frequently recalled (McKeithen, et al., 1981); findings that program text

structured according to these units but with meaningless content is remembered better than randomly arranged lines (Schmidt, 1983); and findings that experienced programmers cluster lines of program text in recall according to procedural unit similarity rather than functional similarity (Adelson, 1981), all support the idea that at least the first stages of comprehension are concerned with a procedural segmentation of text that forms the organizational basis for the programmer's memory representation.

Several alternate views of program comprehension are not supported by the research results. For example, views suggesting strong analogies to chess players' perceptual pattern recognition processes are not supported. Patterns in program text that are recognized quickly appear to be general, few in number, and are discernible through syntactic markers of the language (programming keywords). Thus, a kind of text grammar of limited forms appears to underlie chunking of programming text rather than thousands of content specific patterns that are recognized on presentation (cf. Greeno & Simon, 1984). Proposals that programs are understood in the initial phases of program comprehension through recognition of program plans that assign roles to configurations of program statements is not supported by this research (Rich & Schrobe, 1979; Soloway, et al., 1983). While plan knowledge may be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organizing principles for memory structures. Claims that dataflow relations (Atwood & Ramsey, 1978; Kant & Newell, 1984; Weiser, 1982) or function hierarchies (Adelson, 1984) underlie the preferred or "natural" representation of programs are not supported by this research. Our results suggest that the natural representation of programs is procedural, at least for programs written in traditional programming languages. The best

comprehenders in this study were better at inferring function relations than were poorer comprehenders, but they also showed stronger effects for SP memory organization.

A final result concerns the influence of programming language on comprehension (Green, 1980; Green, et al., 1980). There is an indication in the data that programming language may affect the relationships represented in initial phases of comprehension and the difficulty of extracting different kinds of information from the program. COBOL programmers were consistently better at responding to comprehension questions about dataflow than were FORTRAN programmers. Further, control flow relations were less easily inferred by COBOL programmers. This may be due to features of COBOL and FORTRAN that allow FORTRAN to be programmed in an order corresponding more closely to actual flow of control from statement to statement (at least in these short segments). In COBOL it is more usual to "perform" loops which are then listed elsewhere in the code. Thus the surface structure of the text in COBOL corresponds less well to execution sequence than in FORTRAN. This can be seen by comparing Figure 6.A (program text) and 6.B (SP analysis) in which the loop that sums a list of numbers is executed in statement 5 but is specified in statement 11 through 15. In the FORTRAN version of this segment, the loop occurs in statements 5 through 9 of the code and is executed when encountered there. Thus, there is some evidence that disruption of procedural units in the program text may affect comprehension patterns. Although the data are consistent with a hypothesis that greater difficulty in extracting <u>procedural</u> text units is related to greater difficulty in extracting program <u>function</u>, we cannot firmly conclude that from this study since COBOL and FORTRAN languages and programmers differ in other ways as well.

It is also important to wonder to what extent the particular task used in this research has led to these results. For example, the conclusion that comprehension has a more bottom-up character and is organized in memory by procedural control constructs might be peculiar to understanding small program texts that do not have a larger context. Even if this is the case, it does not dilute the importance of the present research. First, most research on programming skill has used similar short texts. Thus findings in this skill domain must be reconciled with the current results. For example, it has been suggested that expert programmers' recall of texts just like these is due to recognition of program plans (Greeno & Simon, 1984; Soloway, et al., 1983). The current results suggest that this is not the case; that chunking in recall is explained by the grouping of statements into the sequence, loop, and conditional text units suggested by structured programming advocates. Second, the program segments used in this research were all texts that were originally embedded within larger contexts. To the extent that the larger context does not illuminate all program segments equally, the kind of processing reported in this research will certainly occur. Preliminary data from a study of programmers thinking aloud while studying and modifying programs of moderate length (200 lines) in which considerable real world problem context is available is consistent with the data from this study in leading one to believe that the amount of detail and bottom up processing should not be underestimated.

DISCUSSION

At the outset, a larger framework was outlined within which these empirical results may be considered. An argument was presented that a wide range of problem solving tasks, designated as design tasks (Greeno & Simon, 1984), have in common that the goal of problem solving is to create a solution

plan or design that will be interpreted or executed by someone other than the problem solver. A critical element in problem solving in these tasks is the need to simplify the design space in order to make progress in composition and comprehension of designs. These simplifications of necessity foreground some relations in the design and neglect others. A complete design involves the coordination of multiple abstractions describing different kinds of relationships between design elements.

Specific abstractions expressing important relationships in the design of computer programs include a goal hierarchy highlighting major functional achievements of a program (Figure 2), a dataflow abstraction highlighting the transformations that are applied to data objects (Figure 3), a control flow abstraction highlighting the temporal sequence of execution of program actions (Figure 4), and a conditionalized action representation specifying states of the program and the actions invoked (Figure 5). While this particular analysis is specific to problem solvng in computer programming, analogies can be developed for closely related tasks that involve composing other kinds of texts, and more distant analogies for other design tasks in which other kinds of relationships, for example spatial relations, are more central.

The competing theories of computer program comprehension contrasted throughout this report, plan knowledge theory (PK) and structured programming theory (SP), represent claims about which kinds of relationships outlined in the multiple abstractions analysis play a central organizing role in program comprehension: PK theory suggesting that dataflow and function relationships will be dominant and SP theory suggesting that control flow or procedural relationships will be central. More generally though, these two views represent positions about the role of different kinds of knowledge in comprehension. SP theory emphasizes the role of abstact knowledge of program

text structures while PK theory emphasizes the role of a larger body of content schema knowledge that links specific program functions to plans that achieve them.

The present research results come down strongly on the side of SP theoretical formulations: that abstract knowledge of program text structures plays an organizing role in memory for programs, and the control flow or procedural relations dominate in the memory representation. These results are entirely consistent with conclusions reached by researchers in other text comprehension domains who suggest that knowledge of narrative and expositional text structures guides comprehension processing and plays an important role above and beyond other content schematic factors (e.g., Carpenter & Just, 1981; Cirilo & Foss, 1980; Haberlandt, Berian, & Sandson, 1980; van Dijk & Kintsch, 1983; Johnson & Mandler, 1980; Kieras, 1985; Kintsch & van Dijk, 1978; Mandler, 1978, 1984; Mandler & Johnson, 1977; Rumelhart, 1975, 1980; Stein & Glenn, 1979; Thorndyke, 1977). These results are not consistent with conclusions suggesting such knowledge is not involved in comprehension in domains where extensive content knowledge may be available (e.g., Black & Bower, 1980; Black & Wilensky, 1979; Bruce, 1980; Schank & Abelson, 1977; Thorndyke & Yekovich, 1980). Thus, as there is good evidence that "episodes" function as psychological units in story comprehension, there is also good evidence that structured programming building blocks function as psychological units in program comprehension.

In terms of the multiple abstractions analysis, programmers' mental representations in this research were closest to the procedural representation (Figure 4) based on control flow relationships. Should we then conclude that a procedural form is the "natural" mental representation? In the current research, it was originally expected that mental representations would show

function and dataflow relations to be primary. If that had occured, then there would be ample ground to claim that these relations reflected a "natural" or preferred cognitive organization because text and language structure as well as the programmers' training conspire to highlight procedural relations. However, with the current results, we do not know whether the mental organization reflects language/text structure and training, or cognitive "naturalness." There is some evidence from research on understanding instructions that the memory structure reflects procedural relations rather than functional relations whether or not the text from which the procedure is learned has a procedural form (Smith & Spoehr, 1984). On the other hand, the small language differences found in this research suggest that language structure will matter, that the form in which it is convenient to represent a design will be a form that is more closely related to the structure of the stimulus than to the organization of content knowledge in the domain. This is consistent with an emphasis that was popular in earlier times in problem solving research using naive problem solvers and well-structured problems: that stimulus structures provide a major source of influence in the form of mental representations, even for logically isomorphic problems (Hayes & Simon, 1977). Since that time, emphasis has shifted, particularly in research in expert skill domains, to focus on the contributions of content schematic knowledge. While the present results do not suggest that mental representations "reside" solely in the stimulus, they imply that stimulus structures are very important in expert skill tasks.

REFERENCES

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. Memory and Cognition, 9, 422-433.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. Journal of Experimental Psychology: Learning, Memory, and Cognition, 10, 483-495.

Anderson, J. R. (1980). Cognitive psychology and its implications. San Francisco: W. H. Freeman and Co.

Anderson, J. R. (1983). The architecture of cognition. Cambridge, Mass: Harvard University Press.

Atwood, M. E., & Jeffries, R. (1980). Studies in plan construction I: Analysis of an extended protocol. Technical Report SAI-80-028-DEN, Science Applications, Inc., Englewood, Colorado.

Atwood, M. E., & Ramsey, H. R. (1978). Cognitive structures in the comprehension and memory of computer programs: An investigation of computer program debugging. Technical Report SAI-78-054-DEN, Science Applications, Inc., Englewood, Colorado.

Black, J. B., & Bower, G. H. (1980). Story understanding as problem-solving. Poetics, 9, 223-250.

Black, J. B., & Wilensky, R. (1979). An evaluation of story grammars. Cognitive Science, 3, 213-230.

Brooks, R. E. (1975). A model of human cognitive behavior in writing code for computer programs. Doctoral dissertation, Carnegie-Mellon University.

Brooks, R. E. (1983). Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, 18, 543-554.

Bruce, B. (1980). Analysis of interacting plans as a guide to the understanding of memory structure. Poetics, 9, 295-312.

Carpenter, P. A., & Just, M. A. (1981). Cognitive processes in reading: Models based on readers' eye fixations. In A. M. Lesgold and C. A. Perfetti (Eds.), Interactive processes in reading. Hillsdale, N. J.: Lawrence Erlbaum Associates.

Charness, N. (1976). Memory for chess positions: Resistance to interference. Journal of Experimental Psychology: Human Learning and Memory, 2, 641-653.

Chase, W. G., & Simon, H. A. (1973a). The mind's eye in chess. In W. G. Chase (Ed.), Visual information processing. New York: Academic Press.

Chase, W. G., & Simon, H. A. (1973b). Perception in chess. Cognitive Psychology, 4, 55-81.

Cirilo, R. K., & Foss, D. J. (1980). Text structure and reading time for sentences. Journal of Verbal Learning and Verbal Behavior, 19, 96-109.

Curtis, B., Forman, I., Brooks, R. E., Soloway, E., & Ehrlich, K. (1984). Psychological perspectives for software science. Information Processing and Management, 20, 81-96.

Davis, J. S. (1984). Chunks: A basis for complexity measurement. Information Processing and Management, 20, 119-127.

van Dijk, T. A., & Kintsch, W. (1983). Strategies of discourse comprehension. New York: Academic Press.

Egan, D. E., & Schwartz, B. J. (1979). Chunking in recall of symbolic drawings. Memory and Cognition, 7, 149-158.

Engle, R. W., & Bukstel, L. (1978). Memory processes among bridge players of differing expertise. American Journal of Psychology, 91, 673-689.

Flower, L. S., & Hayes, J. R. (1980). The dynamics of composing: Making plans and juggling constraints. In L. W. Gregg and E. R. Steinberg (Eds.), Cognitive processes in writing. Hillsdale, N. J.: Lawrence Erlbaum Associates.

Fodor, J. A., Bever, T. G., & Garret, M. F. (1974). The psychology of language. New York: McGraw-Hill.

Glucksberg, S., & McCloskey, M. (1981). Decisions about ignorance: Knowing that you don't know. Journal of Experimental Psychology: Human Learning and Memory, 7, 311-325.

Goldberg, A., Robson, D., & Ingalls, D. H. H. (1981). Smalltalk-80: The language and its implementation.

Green, T. R. G. (1980). Programming as a cognitive activity. In H. T. Smith and T. R. G. Green (Eds.), Human interaction with computers. New York: Academic Press.

Green, T. R. G., Sime, M. E., & Fitter, M. J. (1980). The problems the programmer faces. Ergonomics, 23, 893-907.

Greeno, J. G., & Simon, H. A. (1984). Problem solving and reasoning. Technical report UPITT/LRDC/ONR/APS-14, University of Pittsburgh.

de Groot, A. D. (1965). Thought and choice in chess. Paris: Mouton and Company.

Haberlandt, K. (1980). Story grammar and reading time of story constituents. Poetics, 9, 99-118.

Haberlandt, K., Berian, C., & Sandson, J. (1980). The episode schema in story processing. Journal of Verbal Learning and Verbal Behavior, 19, 635-650.

Halpern, A. R., & Bower, G. H. (1982). Musical expertise and melodic structure in memory for musical notation. American Journal of Psychology, 95, 31-50.

Hayes, J. R., & Simon, H. A. (1977). Psychological differences among problem isomorphs. In N. J. Castellan, D. B. Pisoni, and G. R. Potts (Eds.), Cognitive theory, Volume 2. Hillsdale, N. J.: Lawrence Erlbaum Associates.

Hayes-Roth, B., & Hayes-Roth, F. (1979). A cognitive model of planning. Cognitive Science, 3, 275-310.

Jackson, M. A. (1980). The design and use of conventional programming languages. In H. T. Smith and T. R. G. Green (Eds.), Human interaction with computers. London: Academic Press.

Jeffries, R., Turner, A. A., Polson, P. G., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), Cognitive skills and their acquisition. Hillsdale, N. J.: Lawrence Erlbaum Associates.

Johnson, N. S., & Mandler, J. M. (1980). A tale of two structures: Underlying and surface forms in stories. Poetics, 9, 51-86.

Kant, E., & Newell, A. (1984). Problem solving techniques for the design of algorithms. Information Processing & Management, 20, 97-118.

Kieras, D. (1985). Thematic processes in the comprehension of technical prose. In B. K. Britton and J. B. Black (Eds.), Understanding expository text. Hillsdale, N. J.: Lawrence Erlbaum Associates.

Kintsch, W. & van Dijk, T. A. (1978). Toward a model of text comprehension and production. Psychological Review, 85, 363-394.

Linger, R. C., Mills, H. D., & Witt, B. I. (1979). Structured programming: Theory and practice. Reading, Mass.: Addison-Wesley Publishing Co.

Mandler, J. M. (1978). A code in the node: The use of a story schema in retrieval. Discourse Processes, 1, 14-35.

Mandler, J. M. (1984). Stories, scripts and scenes: Aspects of schema theory. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Mandler, J. M., & Goodman, M. S. (1982). On the psychological validity of story structure. Journal of Verbal Learning and Verbal Behavior, 21, 507-523.

Mandler, J. M., & Johnson, N. S. (1977). Remembrance of things parsed: Story structure and recall. Cognitive Psychology, 9, 111-151.

Mayer, R. E. (1977). A psychology of learning BASIC. Communications of the ACM, 22, 589-593.

McKeithen, K. B., Reitman, J. S., Reuter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. Cognitive Psychology, 13, 307-325.

McKoon, G., & Ratcliff, R. (1980). Priming in item recognition: The organization of propositions in memory for text. Journal of Verbal Learning and Verbal Behavior, 19, 369-386.

McKoon, G., & Ratcliff, R. (1984). Priming and on-line text comprehension. In D. E. Kieras and M. A. Just (Eds.), New methods in reading comprehension research. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Mitchell, D. C., & Green, D. W. (1978). The effects of context and content on immediate processing in reading. Quarterly Journal of Experimental Psychology, 30, 609-636.

Molzberger, P. (1983). Aesthetics and programming. Proceedings of the Conference on Human Factors in Computing Systems, Boston, Massachusetts.

Newell, A. & Simon, H. A. (1972). Human problem solving. New York: Prentice-Hall.

Norcio, A. F., & Kerst, S. M. (1983). Human memory organization for computer programs. Journal of the American Society for Information Science, 34, 109-114.

Pennington, N. (1982). Cognitive components of expertise in computer programming: A review of the literature. Technical Report, University of Chicago.

Ratcliff, R., & McKoon, G. (1978). Priming in item recognition: Evidence for the propositional structure of sentences. Journal of Verbal Learning and Verbal Behavior, 17, 403-417.

Reder, L. M. (1982). Plausibility judgments versus fact retrieval: Alternative strategies for sentence verification. Psychological Review, 89, 250-280.

Reitman, J. S. (1976). Skilled perception in GO: Deducing memory structures from inter-response times. Cognitive Psychology, 8, 336-356.

Reitman, W. R. (1965). Cognition and thought. New York: John Wiley & Sons.

Rich, C. (1980). Inspection methods in programming. Technical report #604, Massachusetts Institute of Technology Artificial Intelligence Laboratory.

Rich, C. (1981). A formal representation for plans in the programmer's apprentice. Proceedings of the 7th International Joint Conference on Artificial Intelligence, Vancouver, Canada.

Rich, C., & Schrobe, H. E. (1979). Design of a programmer's apprentice. In Artificial intelligence: An MIT perspective. Cambridge, Mass.: The MIT Press.

Rich, C., & Waters, R. C. (1981). Abstraction, inspection and debugging in programming. Memo #634, Massachusetts Institute of Technology Artificial Intelligence Laboratory.

Rumelhart, D. E. (1975). Notes on a schema for stories. In D. G. Bobrow and A. Collins (Eds.), Representation and understanding: Studies in cognitive science. New York: Academic Press.

Rumelhart, D. E. (1980). A reply to Black and Wilensky. Cognitive Science, 4, 313-316.

Sacerdoti, E. D. (1977). A structure for plans and behavior. New York: Elsevier North-Holland, Inc.

Schank, R. C., & Abelson, R. B. (1977). Scripts, plans, goals, and understanding. Hillsdale, N. J.: Lawrence Erlbaum Associates.

Schmidt, A. L. (1983). Comprehension of computer programs by expert and novice programmers. Doctoral dissertation, Southern Illinois University.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. International Journal of Computer and Information Science, 5, 123-143.

Shneiderman, B. (1980). Software psychology. Cambridge, Mass.: Winthrop Publishers, Inc.

Shrobe, H. E. (1979). Dependency directed reasoning for complex program understanding. Technical report # 503, Massachusetts Institute of Technology Artificial Intelligence Laboratory.

Sloboda, J. A. (1976). Visual perception of musical notation: Registering pitch symbols in memory. Quarterly Journal of Experimental Psychology, 28, 1-16.

Smith, E. E., & Spoehr, K. (1984). Comprehension of instructions for operating devices. Paper presented at ONR Conference, Tucson, Arizona.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, 10, xx-xx.

Soloway, E., Ehrlich, K., & Black, J. B. (1983). Beyond numbers: Don't ask "how many"...ask "why". Proceedings of the Conference on Human Factors in Computer Systems, Boston, Massachusetts.

Soloway. E., Ehrlich, K., & Bonar, Jr. (1982). Tapping into tacit programming knowledge. Proceedings of the Conference on Human Factors in Computer Systems. Gaithersburg, Maryland.

Stein, N.L., & Glenn, C.G. (1979). An analysis of story comprehension in elementary school children. In R. Freedle (Ed.), New directions in Discourse Processing, Volume 2. Norwood, N.J.: Ablex.

Tejirian, E. (1968). Syntactic and semantic structure in the recall of orders of approximation to English. Journal of Verbal Learning and Verbal Behavior, 7, 1010-1015.

Thorndyke, P.W. (1977). Cognitive structures in comprehension and memory of narrative discourse. Cognitive Psychology, 9, 77-110.

Thorndyke, P.W., & Yekovich, F.R. (1980). A critique of schemata as a theory of human story memory. Poetics, 9, 23-50.

Trabasso, T., Secco, T., & van den Broeck, P. (1982). Causal cohesion and story coherence. In H. Mandl, N.L. Stein, and T. Trabasso (Eds.), Learning and comprehension of text. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Waters, R.C. (1979). A method for analyzing loop programs. IEEE Transactions on Software Engineering, 5, 237-247.

Weiser, M. (1982). Programmers use slices when debugging. Communications of the ACM, 25, 446-452.

Wilensky, R. (1983). Planning and understanding. Reading, Mass.: Addison-Wesley Publishing Co.

FOOTNOTES

[1] For complex tasks, designing and planning may be indistinguishable in practice although there is not complete overlap between the two concepts. Planning can occur in a task for a "few steps ahead" without creating a complete solution plan. Design in simple tasks could occur without planning. The distinction is not critical to this paper.

[2] Newer high-level languages are sometimes called non-procedural languages to reflect the fact that the instructions may be written in terms of what is to be accomplished rather than how. However, they should be viewed as languages that require less detail in procedural specification rather than having eliminated it. Thus, I argue that the following analysis is relevant to most languages since the "what" at one level is always the "how" for the next level up.

[3] Contention over the number of conceptual units revolves around whether variations on looping structures should be recognized as separate constructs. These controversies do not affect the following discussion.

[4] It can be less since the programmer may know that a single command in the programming language executes one or more actions.

[5] Data from 6 additional subjects were discarded due to programmer difficulty with English (1), mechanical problems during the course of the experiment resulting in incomplete data (3), and motivational problems in completing the experiment (2).

[6] To some extent these predictions are dependent on the time course of comprehension so that more errors would be made earlier about less accessible information. Analyses of comprehension questions by presentation position

were not informative due to the small number of items per cell at this level of analysis.

[7] Response times were standardized for each subject; the response time predicted by the subject's syllables/response time correlation was subtracted out; the remainder is a standardized residual "due to thinking." In any case, no differences in results occur using other common methods of adjustment.

[8] This does not preclude priming due to some other unknown form of relatedness that coincidently was confounded with SP unit membership. However, this "other" basis would also have to account for converging comprehension question results.

TABLE 1
BACKGROUND PROFILES FOR FORTRAN AND COBOL PROGRAMMERS

	FORTRAN	COBOL
MEAN AGE IN YEARS (range)	36.7 (22-57)	34.8 (20-58)
GENDER - % MALE	95.0%	77.5%
MEAN EDUCATIONAL LEVEL 1=high school 2=post high schoo 3=college degree 4=graduate work	3.4	2.8*
% WITH COLLEGE DEGREE OR MORE	87.5%	57.5%*
MAJOR FOR POST HIGH SCHOOL 1=computer science 2=physical science 3=social science 4=humanities	2.0	2.8*
%MAJORING IN COMPUTER SCIENCE OR PHYSICAL SCIENCE	87 5%	34.3%*
MEAN NUMBER OF OTHER PROGRAMMING LANGUAGES KNOWN (RANGE)	6.0 (0-13)	4.1* (0-15)
MEAN NUMBER OF PROGRAMMING COURSES TAKEN (RANGE)	4.1 (0-14)	3.7 (0-8)
% WHO HAVE TAUGHT PROGRAMMING	42.5%	45.0%
MEAN NUMBER OF YEARS PROGRAMMING (RANGE)	14.5 (4-29)	10.5* (3-25)
MEAN NUMBER OF YEARS AS A PROFESSIONAL PROGRAMMER (RANGE)	10.8 (3-29)	9.5 (3-23)
MEAN NUMBER OF HOURS SPENT CODING, DEBUGGING, MODIFYING PROGRAMS DURING PROFESSIONAL CAREER (RANGE)	12,306 (2,000-38,500)	11,196 (1,000-40,000)

* DIFFERENCE BETWEEN LANGUAGES IS STATISTICALLY RELIABLE, $p < .01$.

TABLE 2
MEAN RESPONSE TIMES FOR TARGET RECOGNITION TEST ITEMS
AS A FUNCTION OF PRIME TYPE

A. ALL SUBJECTS, RESPONSE TIME IN SECONDS

	FORTRAN		COBOL	
	PK PRIME	SP PRIME	PK PRIME	SP PRIME
SUBJECTS GROUP 1	2.691 (A materials)	2.695 (B materials)	2.526 (A materials)	2.834 (B materials)
SUBJECTS GROUP 2	2.248 (B materials)	1.972 (A materials)	3.048 (B materials)	2.594 (A materials)
ALL SUBJECTS	2.470	2.333	2.787	2.714

B. UPPER QUARTILE COMPREHENSION SUBJECTS, RESPONSE TIME IN SECONDS

	FORTRAN		COBOL	
	PK PRIME	SP PRIME	PK PRIME	SP PRIME
SUBJECTS GROUP 1	2.560 (A materials)	2.463 (B materials)	2.667 (A materials)	2.948 (B materials)
SUBJECTS GROUP 2	2.220 (B materials)	1.807 (A materials)	2.780 (B materials)	2.063 (A materials)
ALL SUBJECTS (Q1)	2.391	2.135	2.724	2.505

TABLE 3
COMPREHENSION QUESTION ERROR RATES BY INFORMATION CATEGORY

A. MEAN PERCENT ERRORS FOR ALL SUBJECTS

	LANGUAGE		
INFORMATION CATEGORY	FORTRAN	COBOL	COMBINED
OPERATIONS	11.9	17.9	14.9
CONTROL FLOW	17.1	25.2	21.2
DATAFLOW	30.2	24.8	27.5
STATE	31.0	30.7	30.0
MAJOR FUNCTION	26.7	41.5	34.1
ALL CATEGORIES	23.4	28.0	25.7

B. MEAN PERCENT ERRORS FOR TOP QUARTILE SUBJECTS

	LANGUAGE		
INFORMATION CATEGORY	FORTRAN	COBOL	COMBINED
OPERATIONS	3.6	9.1	6.3
CONTROL FLOW	4.7	19.4	12.0
DATAFLOW	18.3	17.6	17.9
STATE	23.2	23.0	23.1
MAJOR FUNCTION	13.0	19.4	16.2
ALL CATEGORIES	12.5	17.7	15.1

C. MEAN PERCENT ERRORS FOR BOTTOM QUARTILE SUBJECTS

	LANGUAGE		
INFORMATION CATEGORY	FORTRAN	COBOL	COMBINED
OPERATIONS	25.6	29.6	27.6
CONTROL FLOW	29.2	37.4	33.3
DATAFLOW	40.2	33.3	36.8
STATE	40.4	36.4	38.4
MAJOR FUNCTION	42.6	51.1	46.9
ALL CATEGORIES	35.6	37.6	36.6

TABLE 4
MEAN RESPONSE TIMES AND RESIDUAL RESPONSE TIMES FOR CORRECT COMPREHENSION QUESTION BY INFORMATION CATEGORY

A. MEAN RESPONSE TIME IN SECONDS FOR CORRECT COMPREHENSION QUESTIONS

	LANGUAGE		
INFORMATION CATEGORY	FORTRAN	COBOL	COMBINED
OPERATIONS	4.72	4.48	4.60
CONTROL FLOW	6.59	7.52	7.05
DATAFLOW	5.96	6.00	5.98
STATE	7.38	7.30	7.34
MAJOR FUNCTION	5.38	6.74	6.06
ALL CATEGORIES	6.01	6.41	6.21

B. MEAN RESIDUAL RESPONSE TIME BY INFORMATION CATEGORY (STANDARDIZED AND CORRECTED FOR READING TIME)

	LANGUAGE			
INFORMATION CATEGORY	FORTRAN	COBOL	COMBINED	YES QUESTIONS COMBINED
OPERATIONS	-.22	-.39	-.31	-.29
CONTROL FLOW	.15	.17	.16	-.16
DATAFLOW	-.08	-.04	-.06	-.05
STATE	.28	.17	.23	.08
MAJOR FUNCTION	-.13	.02	-.05	-.11
ALL CATEGORIES	.00	.00	.00	-.11

LIST OF FIGURES

Figure 1. A simple programming problem.

Figure 2. Abstraction of function.

Figure 3. Abstraction of dataflow.

Figure 4. Abstraction of control flow.

Figure 5. Abstraction of conditions and actions.

Figure 6. Example analyses of a program segment.

Figure 7. Response times for recognition memory items comparing PK-primed item times to SP-primed item times for each set of materials within language adjusted for the effects of subject group (Panel A) and for each subject group within language adjusted for the effects of materials set (Panel B).

Figure 8. Comprehension question error rates by information category for each language.

Figure 9. Comprehension question error rates by information category for top and bottom quartile subjects within each language.

Figure 10. Comprehension question residual response times (questions answered correctly) by information category for each language.

PROBLEM: REARRANGE A TABLE OF CODES SO THAT ALL TYPE "A" CODES COME BEFORE ALL TYPE "B" CODES IN THE TABLE. LABEL EACH CODE AS TO ITS TYPE. PRINT OUT THE NUMBER OF "A" AND "B" CODES IN THE TABLE.

FIGURE 1. A SIMPLE PROGRAMMING PROBLEM

GOAL HIERARCHY: THE PROGRAM ACCOMPLISHES CERTAIN GOALS BY PRODUCING OUTPUTS. EACH LEVEL INDICATES A HIGHER ORDER GOAL IS DECOMPOSED INTO SUBGOALS.

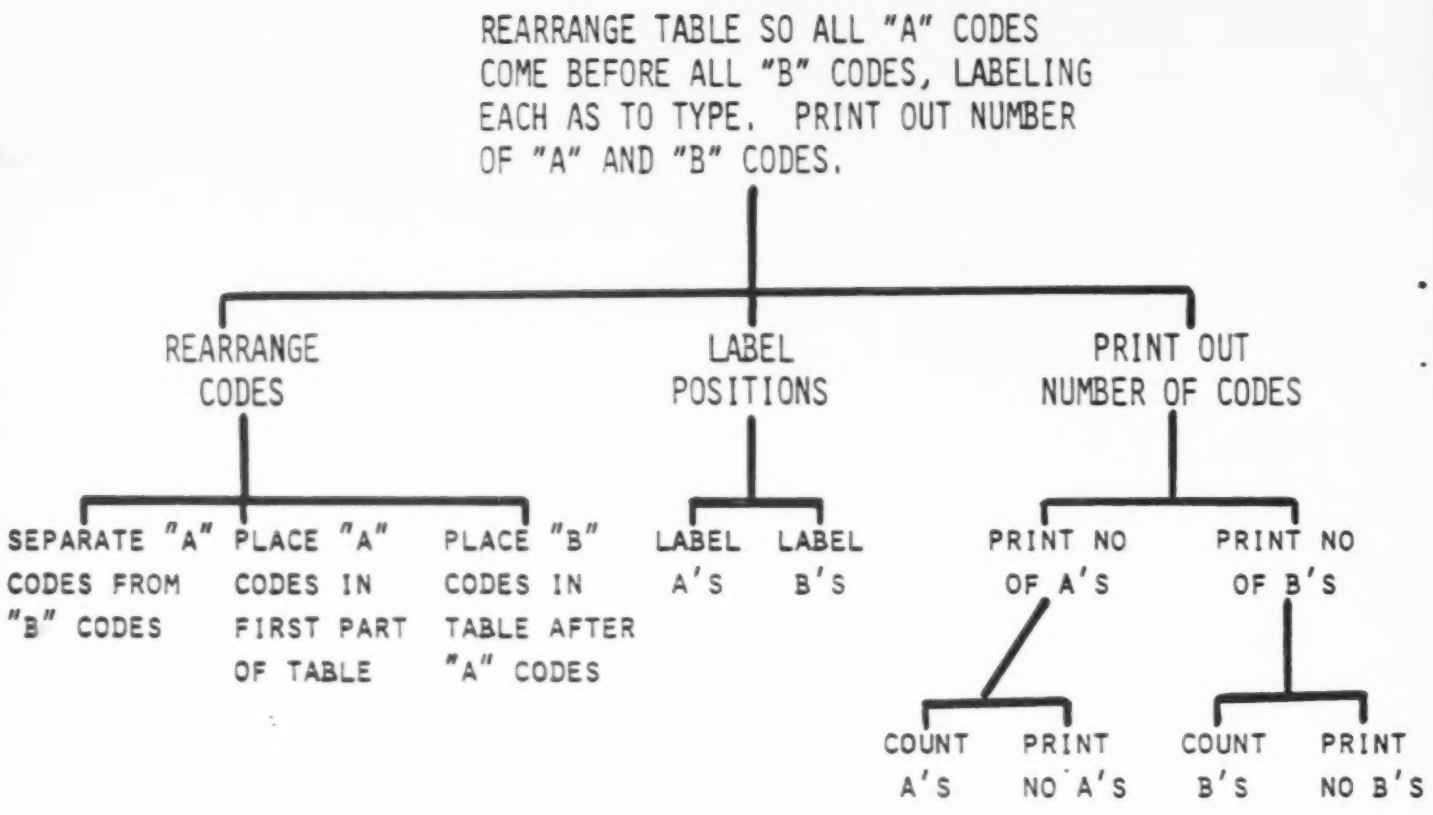

REARRANGE TABLE SO ALL "A" CODES COME BEFORE ALL "B" CODES, LABELING EACH AS TO TYPE. PRINT OUT NUMBER OF "A" AND "B" CODES.
REARRANGE CODES
LABEL POSITIONS
PRINT OUT NUMBER OF CODES
SEPARATE "A" CODES FROM "B" CODES
PLACE "A" CODES IN FIRST PART OF TABLE
PLACE "B" CODES IN TABLE AFTER "A" CODES
LABEL A'S
LABEL B'S
PRINT NO OF A'S
PRINT NO OF B'S
COUNT A'S
PRINT NO A'S
COUNT B'S
PRINT NO B'S

FIGURE 2.

DATAFLOW REPRESENTATION: PROGRAM ACTIONS TRANSFORM INITIAL DATA OBJECTS INTO FINAL DATA OBJECTS. ⬭ INDICATE DATA OBJECTS. ▭ INDICATE PROGRAM ACTIONS.

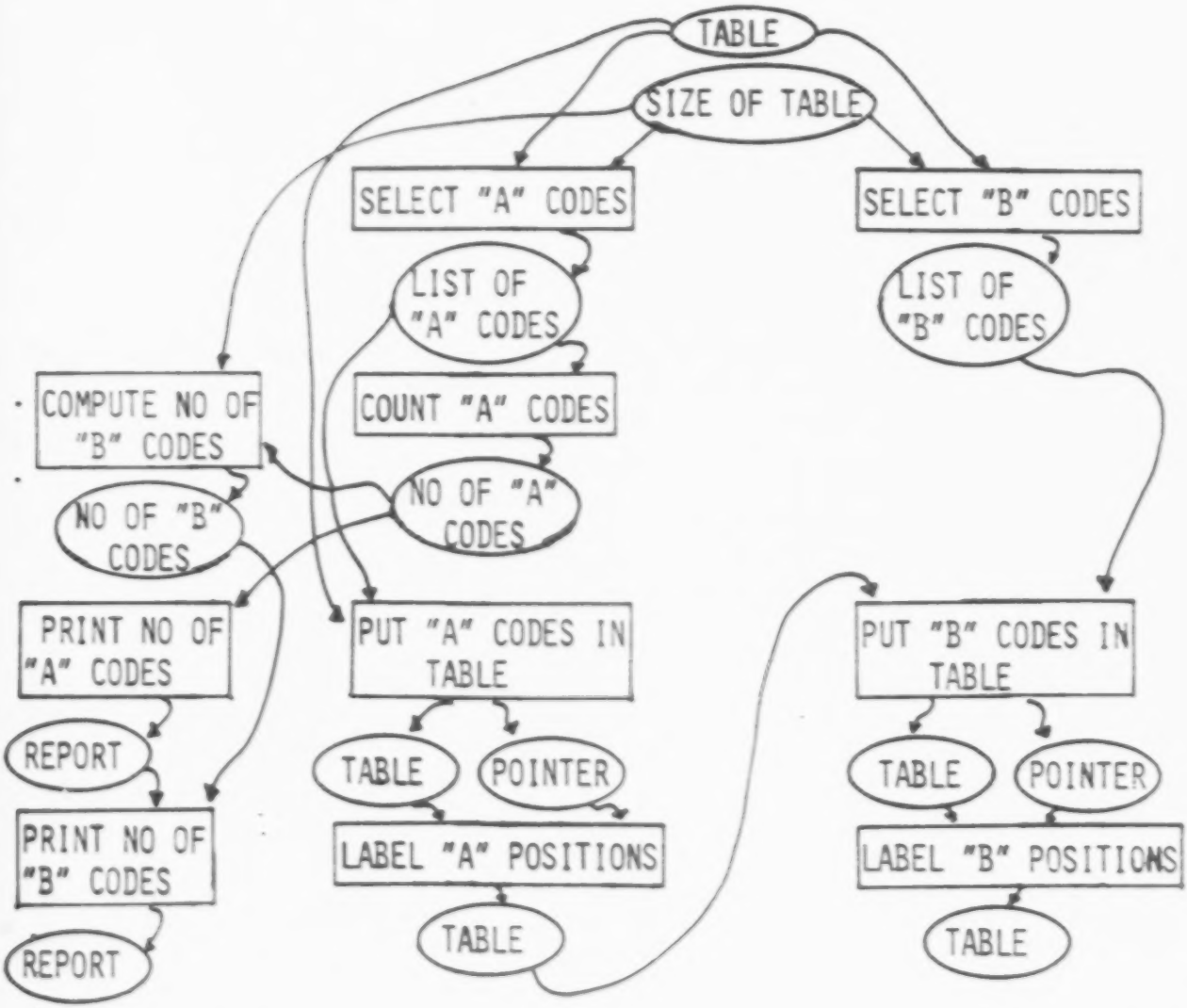


FIGURE 3.

CONTROL FLOW REPRESENTATION: PROGRAM ACTIONS OCCUR IN A SPECIFIED SEQUENCE

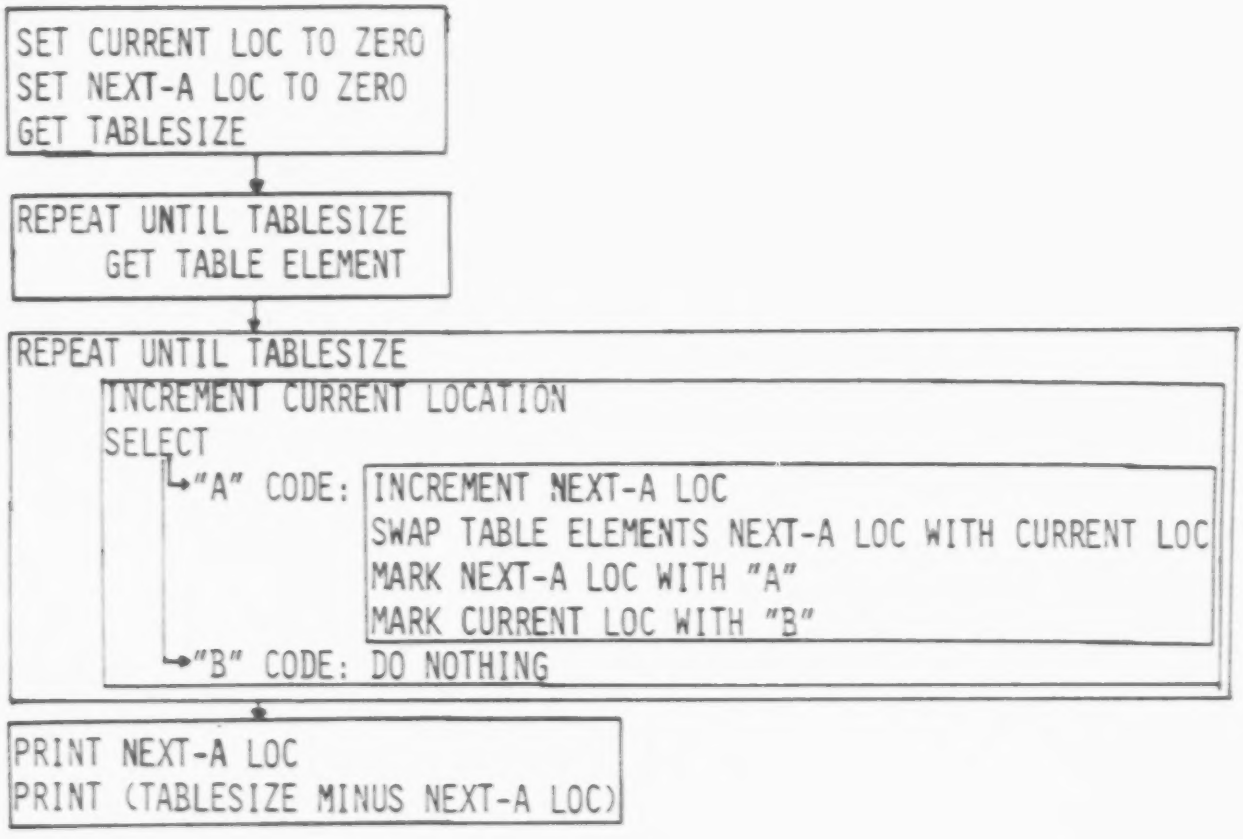


FIGURE 4.

CONDITIONALIZED ACTION REPRESENTATION: A SET OF CONDITIONS RESULTS IN THE EXECUTION OF SOME ACTION(S). THE EXECUTION OF AN ACTION RESULTS IN A NEW SET OF CONDITIONS.

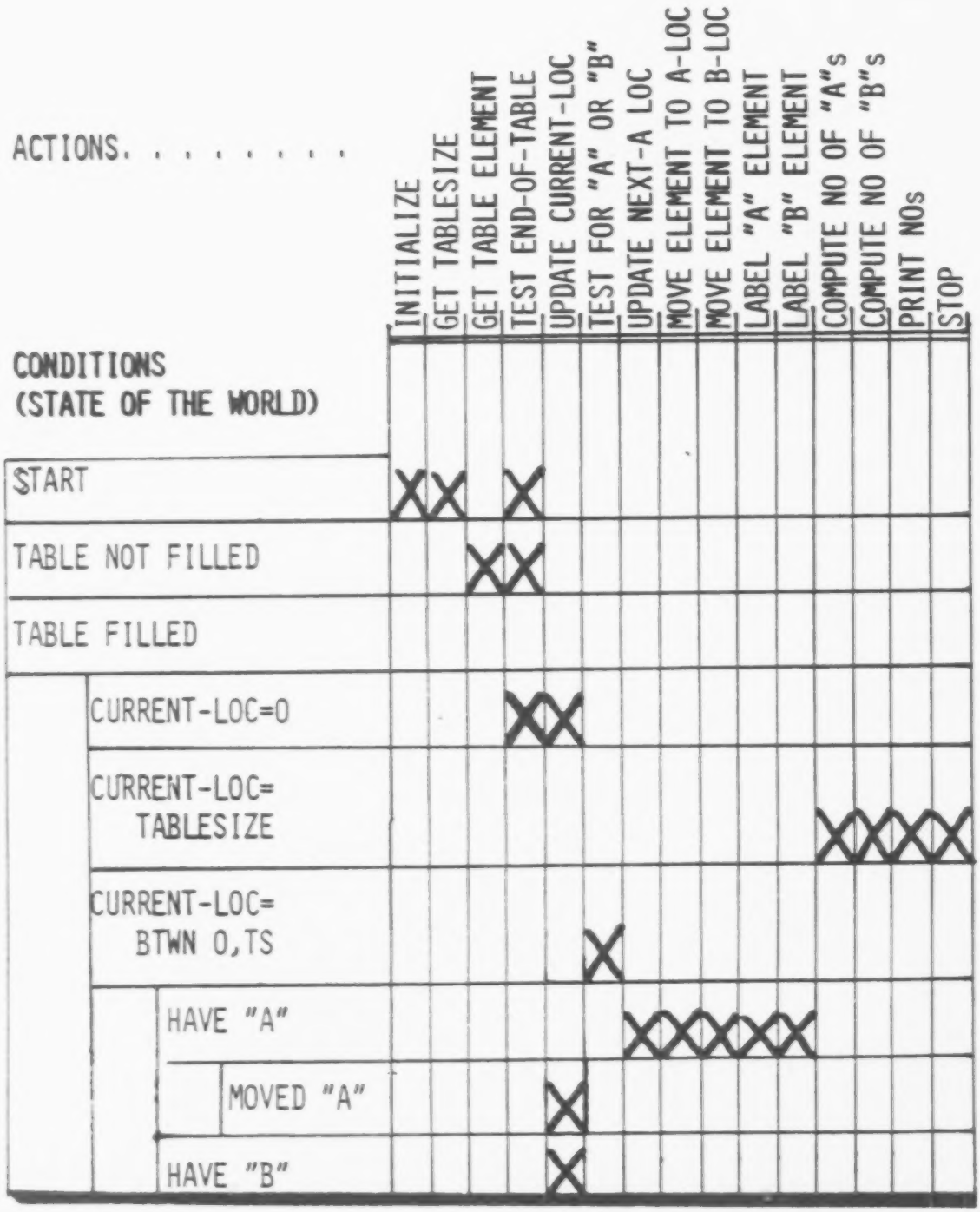


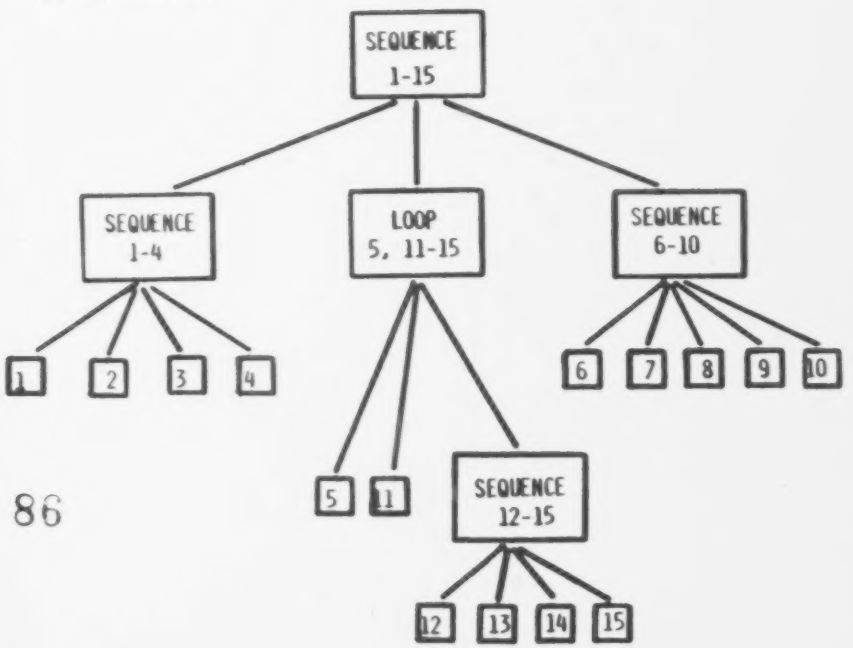
FIGURE 5.

A. COBOL PROGRAM SEGMENT

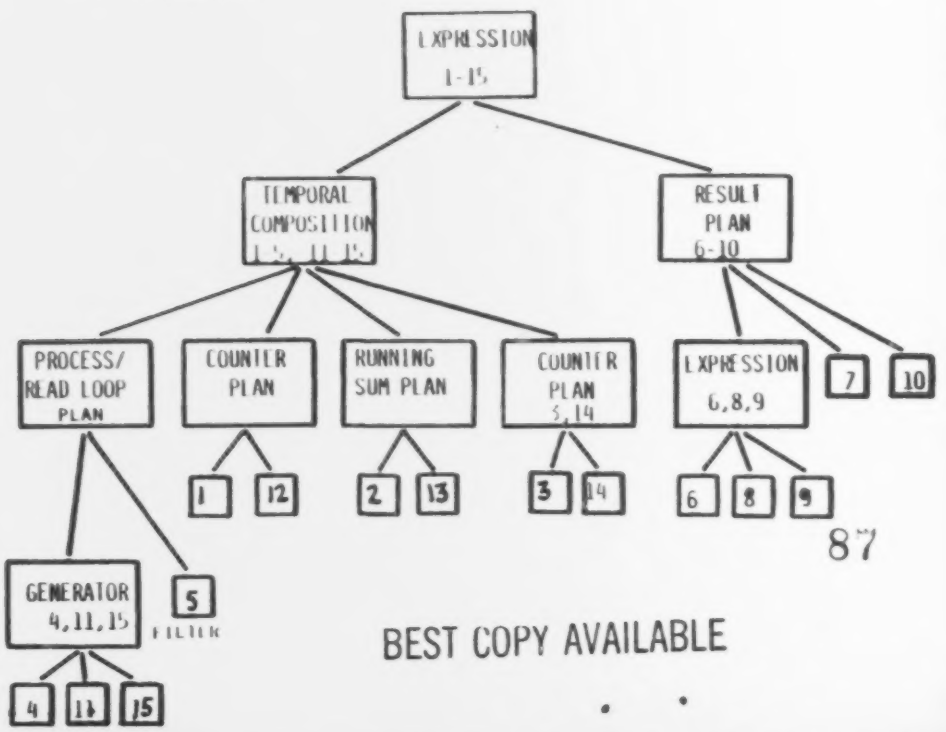
```
STATEMENT
NUMBER

  1          MOVE ZERO TO COUNT-CLIENTS.
  2          MOVE ZERO TO TOTAL-ORDERS.
  3          MOVE ZERO TO INACTIVE-CLIENTS.
  4          READ ORDER-FILE INTO ORDER-REC.
  5          PERFORM SUM-ORDERS UNTIL ORDER-REC-ID = 999999.
  6          COMPUTE ACTIVE-CLIENTS = COUNT-CLIENTS - INACTIVE-CLIENTS.
  7          COMPUTE CLIENT-AVG = TOTAL-ORDERS/COUNT-CLIENTS.
  8          COMPUTE ACTIVE-AVG = TOTAL-ORDERS/ACTIVE-CLIENTS.
  9          DISPLAY MSG-1, ACTIVE-AVG UPON PRINTER.
 10          GO TO ORDER-EXIT.
 11     SUM-ORDERS.
 12          ADD 1 TO COUNT-CLIENTS.
 13          ADD ORDER-REC-QUANT TO TOTAL-ORDERS.
 14          IF ORDER-REC-QUANT = ZERO ADD 1 TO INACTIVE-CLIENTS.
 15          READ ORDER-FILE INTO ORDER-REC.
```

FIGURE 6.

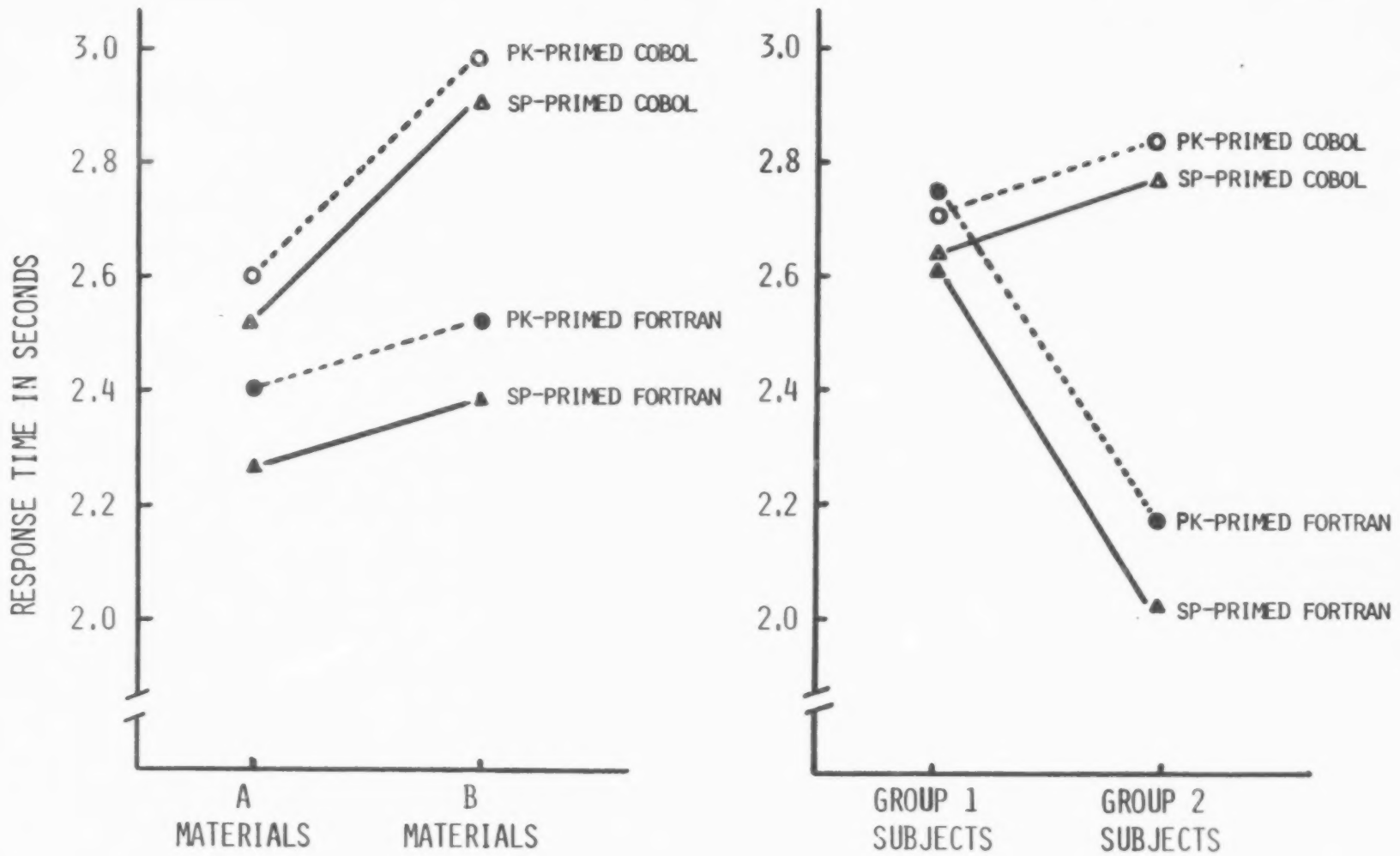
B. SP ANALYSIS



C. PK ANALYSIS



BEST COPY AVAILABLE



A. MATERIALS SETS

B. SUBJECT GROUPS

FIGURE 7.

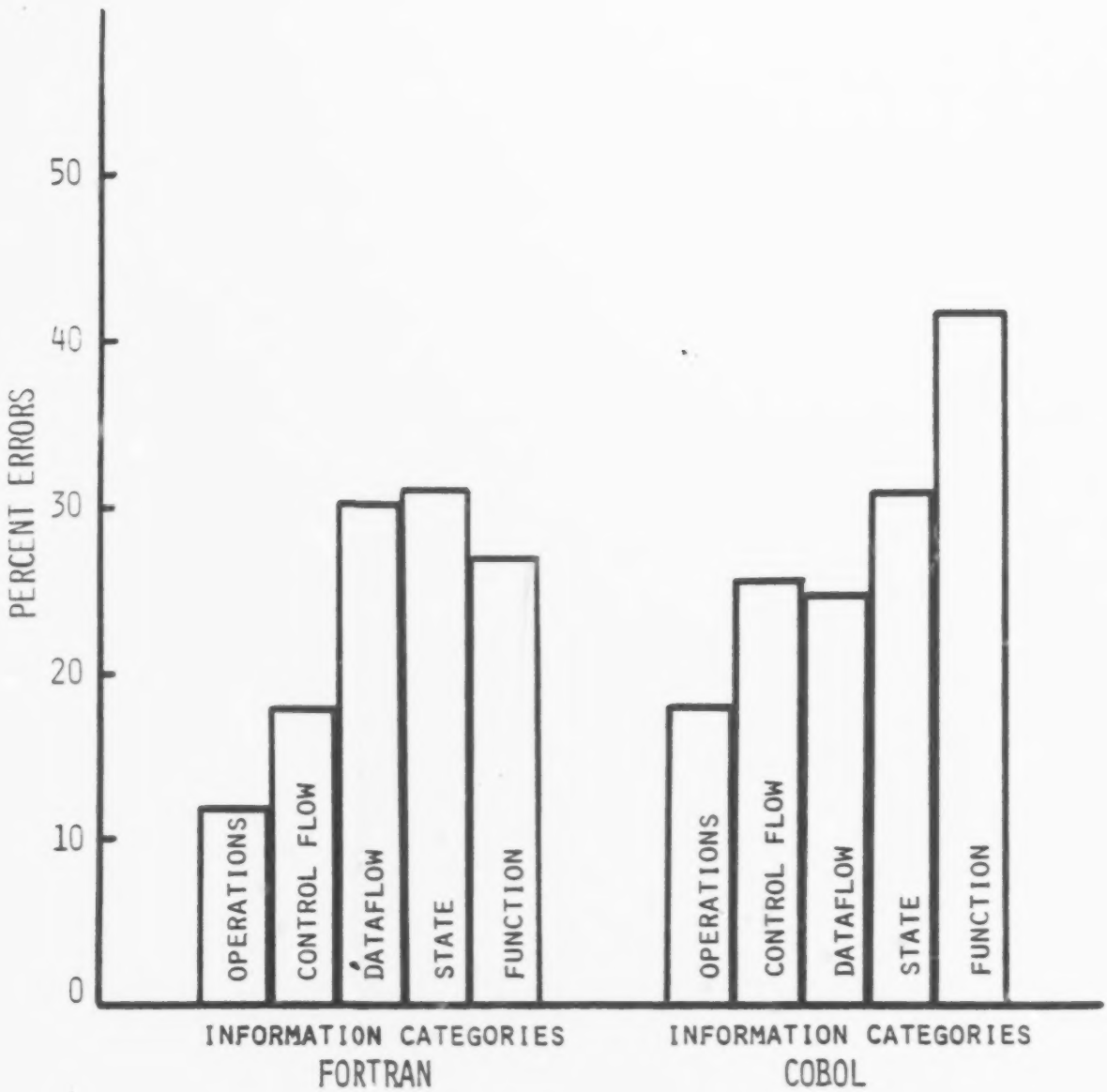

PERCENT ERRORS
50
40
30
20
10
0
OPERATIONS
CONTROL FLOW
DATAFLOW
STATE
FUNCTION
INFORMATION CATEGORIES
FORTRAN
OPERATIONS
CONTROL FLOW
DATAFLOW
STATE
FUNCTION
INFORMATION CATEGORIES
COBOL

FIGURE 8.

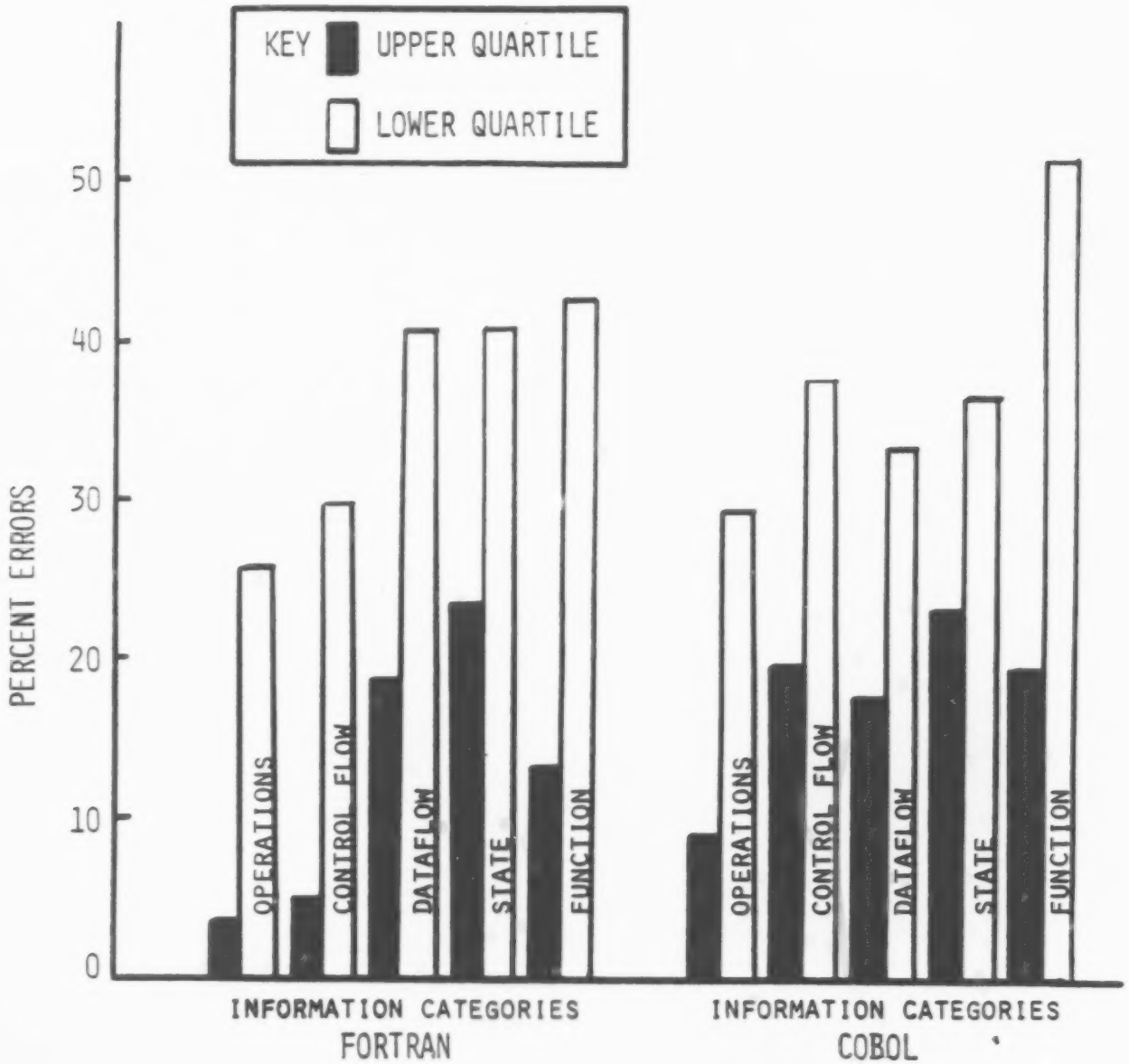
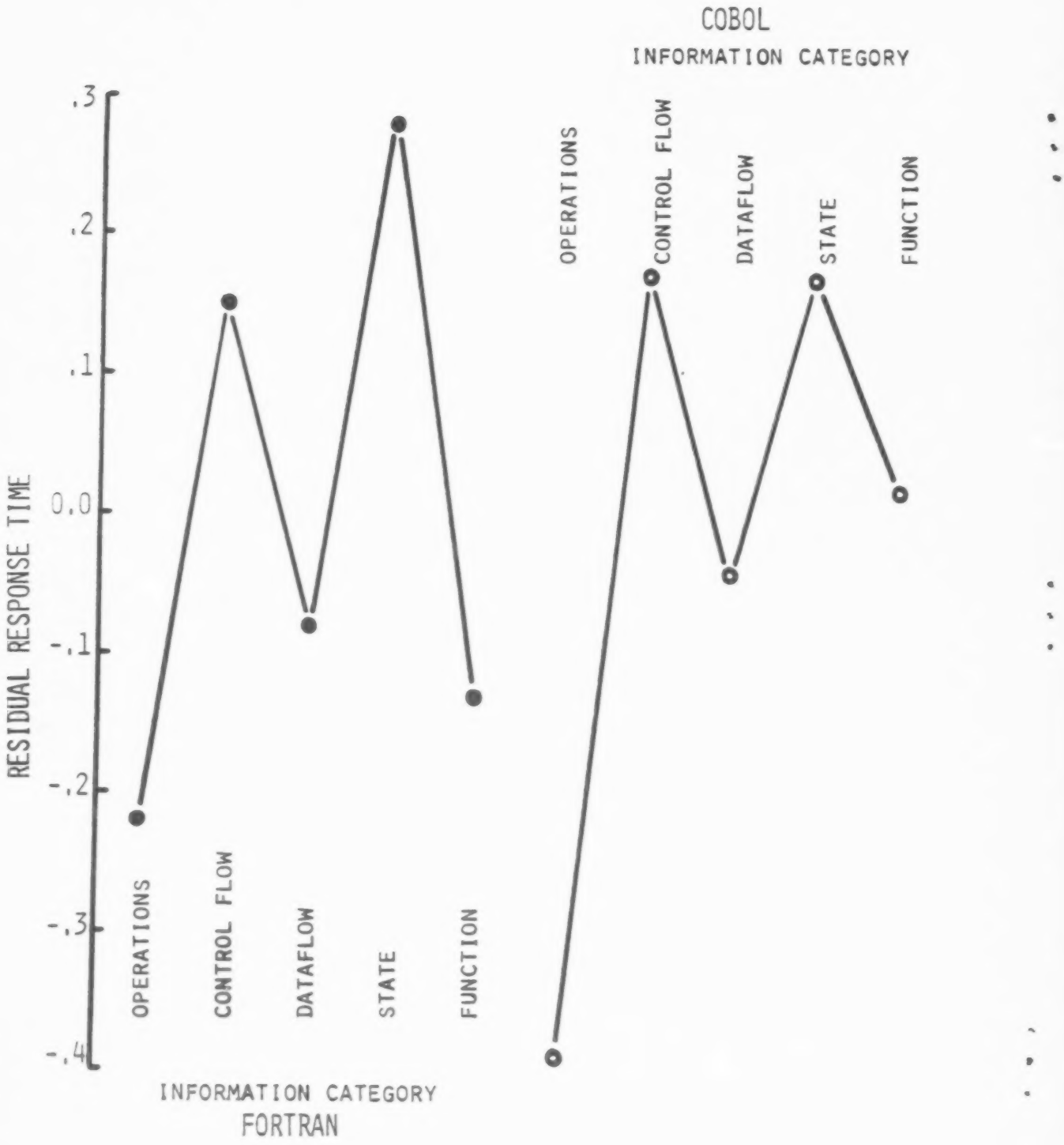

KEY
UPPER QUARTILE
LOWER QUARTILE
PERCENT ERRORS
0
10
20
30
40
50
OPERATIONS
CONTROL FLOW
DATAFLOW
STATE
FUNCTION
INFORMATION CATEGORIES
FORTRAN
INFORMATION CATEGORIES
COBOL

FIGURE 9.



COMPREHENSION QUESTIONS BY LANGUAGE AND CATEGORY

FIGURE 10.